

---

# Parallel Computing

Tobias Lauer

Hochschule Offenburg

---

# Organisatorisches

---

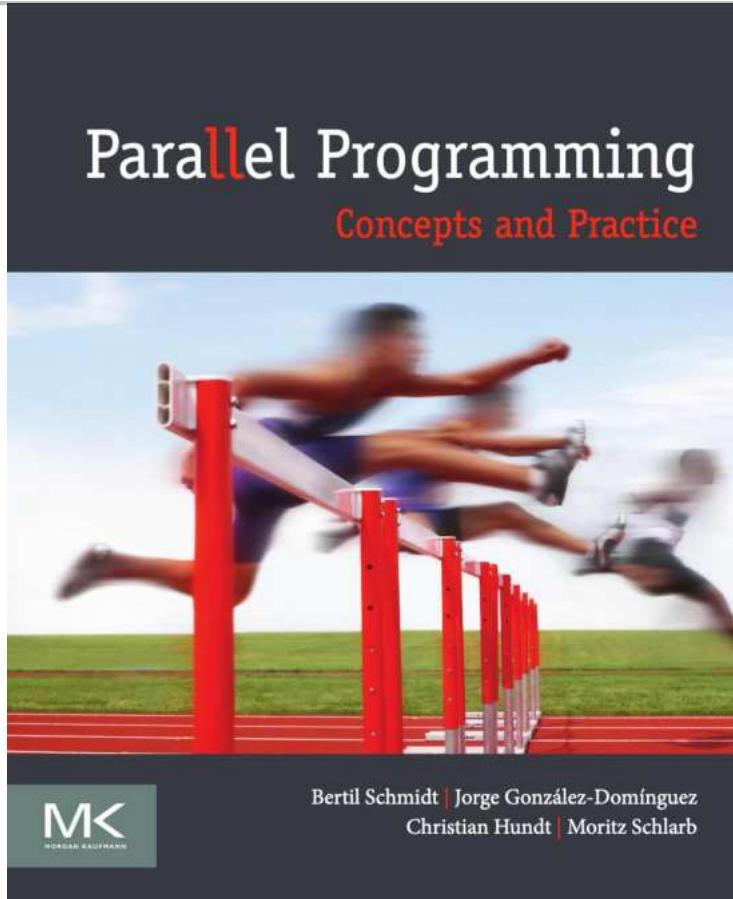
- Ablauf: Vorlesung + Praktikum
  - Termin: *Donnerstag 8:00 - 11:15 Uhr*
  - Aufteilung *Vorlesung/Praktikum unterschiedlich (meistens 1. Block Vorlesung, 2. Block Praktikum)*  
→ wird auf Moodle bekannt gegeben
- Vorlesung:
  - *Folienvortrag*
  - *Freihand-Aufschriebe*
  - *Code-Beispiele*
- Prüfung: *mündlich* (20 min)
  - *Alles o.g. ist prüfungsrelevant*

# Organisatorisches

---

- Praktikum
  - *Übungsaufgaben zum Vorlesungsstoff*
    - Programmierung
    - Verständnis, Analyse, Berechnung
  - *Üblicherweise Abgabe von*
    - Programmcode
    - 1-seitiges „Paper“ zu den Aufgaben
    - i.d.R. 14-täglich
  - *Programmierung in*
    - Java
    - CUDA/C
    - ...

# Literatur



Parallel Programming  
Concepts and Practice

Bertil Schmidt, Jorge González-Domínguez, Christian Hundt, Moritz Schlarb

2018 Elsevier (Morgan Kaufmann)

ca. 61,50 EUR

Sprache: C++ 11

Methoden  
C++11 Multithreading, OpenMP, CUDA, MPI, UPC++

# Literatur



Nebenläufige Programmierung mit Java  
Konzepte und Programmiermodelle für  
Multicore-Systeme

Jörg Hettel, Manh Tien Tran

2016 dpunkt.verlag

ca. 35,50 EUR

Sprache: Java

Methoden  
Multithreading, Threadpools, Atomic,  
Fork-Join, Locks, Sichere Datenstruktu-  
ren, ...

# Themen heute

---

- Motivation
  - *Modellierungsperspektive*
  - *Architekturperspektive*
- Grundbegriffe
  - *Parallele, nebenläufige, verteilte Programme*
- Threads
  - *Bsp. Java*
- Parallelisierung von Algorithmen
  - *2 Beispiele*

# Parallelität in der Informatik

---

- Warum parallele Programmierung?
  - Modellierungsperspektive (Software): „*Die Welt ist parallel*“  
Ziel: *Abbildung dieser Parallelität bei der Modellierung*
  - Architekturperspektive (Hardware): *Prozessorsysteme sind parallel*  
Ziel: *Nutzung von Rechnerressourcen*

# Modellierungsperspektive

---

- Einzelne Applikation
    - *Gleichzeitig abzuarbeitende Teilaufgaben in einer Software, z.B.*
      - Berechnungen, Abfragen
      - User Interface (Darstellung, Interaktion)
    - *Gemeinsamer Zugriff auf Ressourcen*
      - Buchungssysteme
      - Verkehrssysteme
  - Betriebssystem
    - *Zeitgleich abzuarbeitende Prozesse, z.B.*
      - Aktive Anwendungen
      - Hintergrundprozesse (Virenscan etc.)
      - Systemprozesse
- Nebenläufigkeit (englisch: *concurrency*):  
Teile des Systems müssen nebeneinander laufen (können)

# Warum parallele/nebenläufige Programmierung?

---

- Programmstruktur bildet Problemstellung besser ab
- Durchsatz
  - *Weniger Warten z.B. auf I/O Prozesse*
- Response
  - *Interaktionen können mit gewünschter Priorität behandelt werden*
- Steigerung der Ausführungsgeschwindigkeit
  - *Nutzung möglichst aller vorhandenen Rechenressourcen*

# Architekturperspektive: Moore's Law (1960)

---

„Prozessorleistung\* verdoppelt sich ca. alle 18-24 Monate“  
(\*eigentlich: Transistordichte)

→ Neuere Hardware beschleunigt Software automatisch

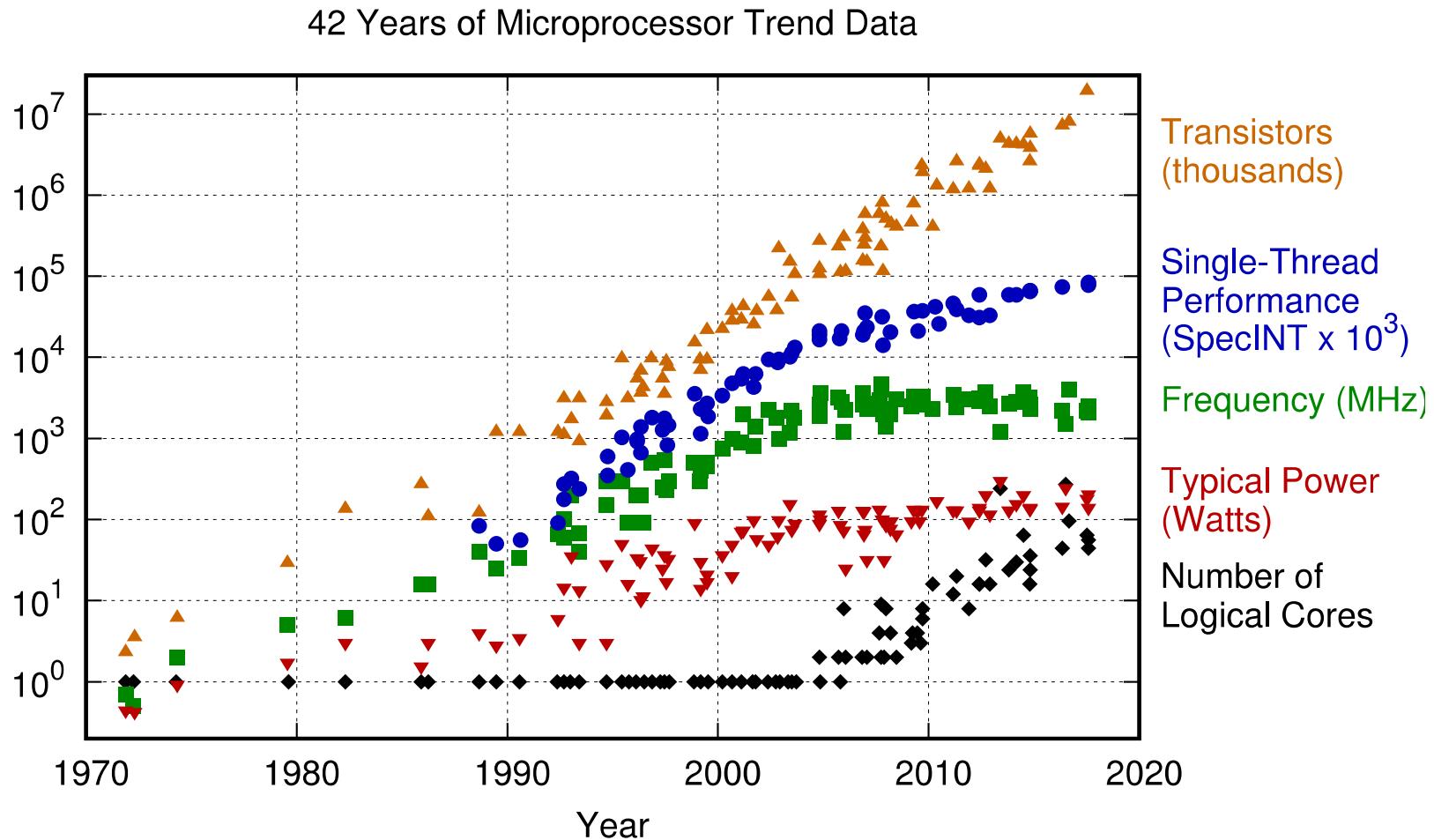
Wirklich?

Seit den 2000er Jahren gilt das nicht mehr ohne Weiteres!

„*The free lunch is over*“ (H. Sutter, 2005)

Zwar weiterhin Erhöhung der Transistorzahl, aber nicht mehr auf *einem* Prozessor(kern)

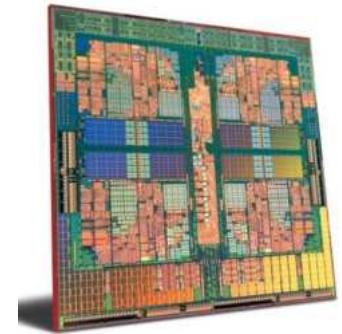
# Moore's Law in Kennzahlen



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2017 by K. Rupp

# Architektur-Perspektive

- Multicore-Prozessoren (4, 8, 16, ... Cores)
  - *Leistungssteigerung benötigt (Moore's Law)*
  - *Grenzen der Miniaturisierung und Höhertaktung*
    - Hitzeentwicklung, Kühlung
    - Prinzipielle physikalische Grenzen
- „Manycore“-Architekturen (> hunderte Cores)
  - *Grafikprozessoren (GPU)*
  - *Tensor Processors (TPU)*
- Computer-Cluster, Supercomputer



# „Multicore-Krise“

---

- Leistungssteigerung ist (zunächst) nur theoretisch
  - *Gut für bereits nebenläufige Systeme (unabhängige Teile werden auf Cores verteilt)*
  - *Aber macht einzelne Anwendungen nicht (mehr) automatisch schneller*
  - *Nicht selten werden sie sogar langsamer!*
- In der Praxis gibt es oft **eine** kritische, ressourcen-hungige Anwendung, die die volle Leistung braucht
  - *Server-Anwendung*
  - *Komplexe/aufwendige Berechnungen*

→ Parallelisierung der **Anwendungen/Berechnungen** nötig!

# Verschärfung: Many-Core

---

- Hunderte oder tausende Cores
  - z.B. moderne Grafikkarten (GPUs), Vektor-/Tensorprozessoren
  - Meist eingeschränkter Befehlssatz
  - Geringere Leistung einzelner Cores
  - Enge physikalische Parallelität  
(SIMD-Modell: „single instruction, multiple data“)  
→ Zwingt zur parallelen Bearbeitung  
(auch von eigentlich sequenziellen Problemen)
- Parallele Algorithmen
  - Parallelisierte Varianten „klassischer“ oder neuer Algorithmen
  - Bausteine („parallel building blocks“)

# Grundbegriffe

---

- **Nebenläufiges** Programm
  - *Mehrere Handlungsfäden*
  - *Logisch gleichzeitig*
  - *Geringe Abhängigkeit*
- (Echt) **paralleles** Programm
  - *Mehrere Prozessoren*
  - *(auch) physikalisch gleichzeitig*
  - *Starke Abhängigkeit (für Gesamtlösung)*
- **Verteiltes** Programm
  - *Mehrere Rechner*
    - Mehrere Prozessoren
    - Mehrere Handlungsfäden
    - Getrennte Ressourcen
- Koordiniere (relativ) unabhängige Aktivitäten
- Versuche, eine gegebene Aktivität aufzuteilen in parallele Teilschritte
- Koordiniere Aktivitäten nur über Nachrichten

# Threads zur Parallelisierung

---

- Unterteilung **einer** Aufgabe in Tasks, die (möglichst) **unabhängig** voneinander und dadurch **gleichzeitig** erledigt werden können
  - Jeder Thread erledigt eine Task (oder mehrere)
  - Die Threads können dabei (hoffentlich) alle vorhandenen Prozessoren/Cores nutzen
- **Beschleunigung** des Programmablaufs möglich

# Beispiel: Array-Count

---

1	0	1	1	1	1	0	0	1	1	0	1	0	1	1	1	1	1	0	1	1	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- Bestimme die Anzahl der Elemente eines Arrays, die eine bestimmte Bedingung erfüllen.
- Oft benötigter algorithmischer Baustein in der Programmierung (z.B. Ausfiltern, Selektieren, etc.)
- Sehr einfach mit  $k$  Prozessoren zu parallelisieren durch Aufteilen des Arrays in  $k$  Teile
- Jeder Teil wird von einem Thread bearbeitet

# Array-Count: Thread

```
class CountThread extends Thread {  
    private boolean[] array;  
    private int lo;  
    private int hi;  
    private int result;  
  
    public CountThread(boolean[] array, int lo, int hi) {  
        this.array = array;  
        this.lo = lo;  
        this.hi = hi;  
    }  
  
    public void run() {  
        for(int i = lo; i <= hi; i++)  
            if(array[i])      // Bedingung  
                result++;  
    }  
  
    public int getResult() {  
        return result;  
    }  
}
```

Übergib Parameter  
im Konstruktor!

Übergib Ergebnis  
mittels Methode!

# Array-Count: Aufteilung

```
public static void main(String[] args) {
    ...

    CountThread[] counter = new CountThread[NUMBER_OF_THREADS];
    int lo = 0;
    int hi;
    int size = ARRAY_SIZE / NUMBER_OF_THREADS;

    for(int i = 0; i < NUMBER_OF_THREADS; i++) {
        if(i < NUMBER_OF_THREADS - 1)
            hi = lo + size - 1;
        else
            hi = ARRAY_SIZE - 1;
        counter[i] = new CountThread(array, lo, hi);
        counter[i].start();
        lo = hi + 1;
    }

    ...
}
```

# Array-Count: Ergebnisse zusammenführen

```
// Synchronisation (auf alle Threads warten)

try {
    for(int i = 0; i < NUMBER_OF_THREADS; i++) {
        counter[i].join();
    }
} catch(InterruptedException e) {
    e.printStackTrace();
}

// Gesamtergebnis aus Teilergebnissen berechnen
int result = 0;
for(int i = 0; i < NUMBER_OF_THREADS; i++) {
    result += counter[i].getResult();
}
```

**join()** führt den Thread, für den es aufgerufen wird, mit dem aufrufenden Thread zusammen.

# Array-Count: Ergebnisse zusammenführen

```
// Synchronisation (auf alle Threads warten)

int result = 0;
try {
    for(int i = 0; i < NUMBER_OF_THREADS; i++) {
        counter[i].join();
        result += counter[i].getResult();
    }
} catch(InterruptedException e) {
    e.printStackTrace();
}
```

# Beschleunigung durch Parallelisierung

---

- Wieviel Beschleunigung kann Parallelisierung bringen?
- Definition der Beschleunigung (Speedup factor)  $S$ :

$$S = t_{\text{original}} / t_{\text{beschleunigt}}$$

- Bei Parallelisierung mit  $p$  Prozessoren:

$$S(p) = t_{\text{sequenziell}} / t_{\text{parallel}}$$

→ Wieviel Speedup ist möglich?

# Speedup bei Array-Count mit $k$ Prozessoren

---

- Array wird in  $k$  Teile zerlegt ( $1 \leq k \leq n$ )
- Je mehr Teilstücke, desto höher die Parallelisierung
  - *Aufwand pro Thread:*  $O(n/k)$
- Aber: je mehr Teilstücke, desto höherer Aufwand beim Zusammenführen der Ergebnisse
  - *Aufwand:*  $O(k)$
- Gesamtaufwand:  $O(n/k) + O(k)$

# Parallele Algorithmen

---

- Bisherige implizite Annahme:
  - $k$  fest vorgegeben (konstant) durch Prozessorzahl im System
- Neuer Blickwinkel:
  - Wir betrachten  $k$  als wählbar in Abhängigkeit der Eingabegröße  $n$
  - Wir nennen  $k = P(n)$  die Prozessorkomplexität des Algorithmus
  - Vorläufige Annahme: „Wir dürfen so viele Prozessoren verwenden, wie wir brauchen“ (um ein optimales Ergebnis zu erzielen)
  - Um den tatsächlichen Speedup auf einem konkreten System mit  $p$  Prozessoren kümmern wir uns später ☺

# Speedup bei Array-Count?

---

- Je mehr Teilstücke, desto höher die Parallelisierung
  - *Aufwand pro Thread:  $O(n/k)$*
- Aber: je mehr Teilstücke, desto höherer Aufwand beim Zusammenführen der Ergebnisse
  - *Aufwand:  $O(k)$*
- Gesamtaufwand:  $O(n/k) + O(k)$ 
  - *Was wäre ein guter/optimaler Wert für  $k$  in Abhängigkeit von  $n$ ?*

# Fortgeschrittenes Beispiel: Editierdistanz

---

## Problem:

Finde für zwei gegebene Strings  $A$  und  $B$  die minimale Anzahl von Editieroperationen, um  $A$  in  $B$  zu überführen.

## Editieroperationen:

- Einfügen eines Zeichens
- Löschen eines Zeichens
- Ersetzen eines Zeichens

# Editierdistanz

Mini-Beispiel:

$$A = \text{SONG} \quad B = \text{HONIG}$$

2 Editieroperationen notwendig/ausreichend:

- Ersetze S durch H: SONG → HONG
- Füge I ein: HONG → HONIG

d.h. Die Editierdistanz  $d(A,B)$  ist 2.

Beobachtungen:

- (1)  $0 \leq d(A,B) \leq \max(|A|, |B|)$
- (2)  $d(A,B) = d(B,A)$

# Anwendungsgebiete

---

- Suchmaschinen
  - *Alternativvorschläge bei Tippfehlern*
- Textverarbeitung
  - *Erkennung von Rechtschreibfehlern*
  - *Automatische Korrektur*
- Plagiatserkennung
  - *Auffinden leicht veränderter Texte*
- Datenbanken
  - *Erkennen von Duplikaten*
- Bioinformatik
  - *Sequence alignment*

# Berechnung der Editierdistanz

---

- **Idee:**  
Wenn die Editierdistanz für Teilwörter (Präfixe) schon bekannt ist, kann daraus die Editierdistanz für längere Teilwörter einfach bestimmt werden.
- Methode der *Dynamischen Programmierung*:  
Ergebnis wird schrittweise aus Teilergebnissen aufgebaut

# Dynamische Programmierung

- Ziel: Berechne eine  $(m+1) \times (n+1)$ -Matrix  $d$  mit den Editierdistanzen aller Präfixkombinationen von  $A$  und  $B$

	H	O	N	I	G
0	1	2	3	4	5
S	1	1	2	3	4
O	2	2	1	2	3
N	3	3	2	1	2
G	4	4	3	2	2

- Editierdistanz von  $A$  und  $B$  findet sich in  $d[m,n]$
- Ermitteln der Editieroperationen durch Backtracking

# Berechnung der Matrix

```
// Berechnet Zelle d[i,j] für i,j > 0
calcCell(i, j)
  if (A[i] == B[j])
    d[i,j] = d[i-1,j-1] // nichts zu tun
  else
    d[i,j] = 1 + min( d[i-1,j], // Einfügung
                           d[i,j-1], // Löschung
                           d[i-1,j-1] ) // Ersetzung
```

	H	O	N	I	G
0	1	2	3	4	5
S	1	1	2	3	4
O	2	2	1	2	3
N	3	3	2	1	2
G	4	4	3	2	2

# Algorithmus

---

```
int m = A.length;
int n = B.length;

int d[][] = new int[m+1][n+1];      // Matrix anlegen

for (int i=0; i<=m; i++)           // Spalte 0 initialisieren
    d[i,0] = i;

for (int j=0; j<=n; j++)           // Zeile 0 initialisieren
    d[0,j] = j;

for (int i=1; i<=m; i++)
    for (int j=1; j<=n; j++)
        calcCell(i,j);

return d[m,n];                      // Gib Ergebnis zurück
```

# Effizienz

---

- Die Laufzeit des Algorithmus liegt in  $O(n \cdot m)$ :
  - Es müssen (im Wesentlichen)  $n \cdot m$  Zellen berechnet werden.
  - Jede einzelne Zelle kann in konstanter Zeit berechnet werden.
  - Die Zellen werden einzeln nacheinander berechnet.
- Bei langen Strings (z.B. in der Bioinformatik) ist das sehr ineffizient
  - Länge der Strings im Millionenbereich und höher
- Gibt es eine Möglichkeit für Parallelisierung,  
d.h. können Zellen gleichzeitig berechnet werden?

# Beobachtungen

---

- Die Reihenfolge der verschachtelten **for**-Schleife ist egal
  - Zeilen- oder spaltenweise Berechnung möglich
- ABER: Berechnung der Zellen in völlig beliebiger Reihenfolge ist nicht möglich
  - Jede Zeile bzw. Spalte benötigt die vorherige Zeile bzw. Spalte.
  - Jede Zeile bzw. Spalte muss auch *in sich schrittweise* berechnet werden.
  - Genauer:  
*Jede Zelle erfordert zur Berechnung u.a. die gerade vorher berechnete Zelle (oberhalb oder links).*

# Parallelisierung

- Andere Berechnungsreihenfolge:
  - *Nicht zeilen- oder spaltenweise, sondern in Diagonalen*
- Alle Zellen auf einer Diagonale können unabhängig voneinander, also parallel, berechnet werden
  - $\begin{matrix} S & A & M & S & T & A & G \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{matrix}$

- ABER: vor jeder neuen Diagonale müssen die vorherigen beiden Diagonalen berechnet sein
  - *Synchronisation notwendig!*

**A**    3    **2**    1

**R**    4    3

# Anzahl und Größe der Diagonalen

	S	A	M	S	T	A	G	
0	1	2	3	4	5	6	7	
S	1	0	1	2	3	4	5	6
T	2	1	1	2	3	3	4	5
A	3	2	1	2	3	4	3	4
R	4	3	2	2	3	4	4	4

```
for (k=1; k<m ; k++)
  for (t=1; t<=k; t++)
    calcCell(k-t+1, t)

for (k=m; k<=n; k++)
  for (t=k-m+1; t<=k; t++)
    calcCell(k-t+1, t)

for (k=n+1; k<n+m; k++)
  for (t=k-m+1; t<=n; t++)
    calcCell(k-t+1, t)
```

# Paralleler Algorithmus (abstrakt)

```
for (k=1; k<n+m; k++) {  
    for t=max(1, k-m+1) to t<=min(k, n) in parallel {  
        calcCell(k-t+1, t)  
    }  
    synchronize  
}
```

Wenig praxistauglich, aber sinnvoll z.B. für Laufzeitabschätzung:

Wenn wir  $m$  Prozessoren haben, wird die innere Schleife in 1 Schritt abgearbeitet (jeder Prozessor berechnet eine Zelle).

Damit haben wir  $n+m-1$  Schritte.

Zur Erinnerung: Der sequentielle Algorithmus benötigt  $n \cdot m$  Schritte.

# Berechnung mit $m$ Threads

	S	A	M	S	T	A	G	
	0	1	2	3	4	5	6	7
S	1	0	1	2	3	4	5	6
T	2	1	1	2	3	3	4	5
A	3	2	1	2	3	4	3	4
R	4	3	2	2	3	4	4	4

Jeder Thread arbeitet genau eine Zeile ab.

```
for (int c=1; c<=n; c++) {  
    if (c>=1 && c<=n)  
        calcCell(id+1, c);  
    // Hier muss eine BARRIERE sein!  
}
```

Jeder Thread arbeitet um 1 Spalte zu seinen Nachbarn versetzt.

Nach jedem Schritt müssen alle Threads synchronisiert werden.

→ Wir brauchen eine Synchronisationsbarriere.

# Die Java-Klasse **CyclicBarrier**

---

- Stellt eine Barriere für eine spezifizierbare Zahl  $m$  von Threads (genauer: Runnables) bereit
- Wenn die Methode `await()` aufgerufen wird, wartet jeder Thread, bis die Barriere aufgehoben wird.
- Dies geschieht, wenn alle  $m$  Threads bei der Barriere angekommen sind (durch internen Countdown).
- Optional kann ein Runnable immer direkt vor Aufhebung der Barriere ausgeführt werden (z.B. für sequentielle Zwischenschritte im Hauptprogramm).

# Java-Rahmen mit Barriere

```
class EditDistance {
    final int m,n;
    final int[][] d;
    final CyclicBarrier barrier;

    public EditDistance(char[] A, char[] B) {
        m = A.length;
        n = B.length;
        d = new int[m+1][n+1];
        barrier = new CyclicBarrier( m,
                                    new Runnable() {
                                        public void run() {
                                            // optionale Aktionen
                                        }
                                    } );
        initMatrix();
        for (int i=0; i<m; i++) {
            new Thread(new CalcThread(i)).start();
        }
    }
}
```

# Benutzung in einem Thread

```
class CalcThread implements Runnable {
    int id;      // 0 <= id < m

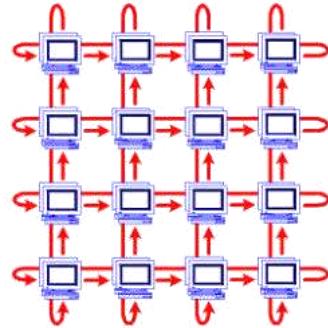
    CalcThread(int threadID) {
        id = threadID;
    }

    public void run() {
        for (int c=1-id; c<n+m-id; c++) {
            if (c>=1 && c<=n)
                calcCell(id+1, c);
            try {
                barrier.await();
            } catch(...) { };
        }
    }
}
```

# Was bringt die Parallelisierung?

---

- Wenn wir  $m$  Prozessoren haben, wird jede Diagonale in einem einzigen Schritt abgearbeitet.
  - Damit haben wir insgesamt  $n+m-1$  Schritte.
  - Zur Erinnerung:  
*Der sequentielle Algorithmus benötigt  $n \cdot m$  Schritte.*
    - Bei  $n=20, m=16$  auf einem 16-Core-Rechner:  
320 vs. 35 Schritte
    - Bei  $n=800, m=768$  auf einer GPU (NVIDIA GTX-1050):  
614.400 vs. 1567 Schritte.
- Wenn die Zahl der Prozessoren  $p < m$  ist, ist die Schrittzahl  $(n+m-1) \cdot m/p$
- Tatsächlicher Speedup hängt von weiteren Faktoren ab.



---

# Parallel Computing

Tobias Lauer  
Hochschule Offenburg

---

# Themen

---

- Perfekte Parallelisierbarkeit
  - *Bsp.: Monte Carlo Simulation*
- Threading in Java
- Threadsicherheit von Objekten
- Synchronisation von Threads
  - *Sprachmittel von Java*
  - *Bibliotheken*

# Parallelisierung

---

- Es gibt Probleme, die sich trivial parallelisieren lassen:
  - *Völlig unabhängige Schritte*
  - *Keine Kommunikation nötig*
  - *Keine (bzw. kaum) Synchronisation*
  - *Beispiele:*
    - Array-Count (s. Vorlesung 1)
    - Grafik-Rendering
    - Viele Probleme der Bildverarbeitung
    - Monte Carlo Simulationen
  - „*embarrassingly parallel*“, „*perfectly parallel*“

# Monte Carlo Simulation

---

- Geeignet für:
  - *Probleme mit unbekannter exakter Lösung*
    - Finanzwelt
    - Physik, Chemie
    - Wahrscheinlichkeiten
  - *Prozesse/Abläufe lassen sich gut simulieren*
- Idee:
  - *Lasse möglichst viele Simulationen laufen*  
→ *Gesetz der großen Zahlen*
  - *Einzelne Simulationen sind komplett unabhängig*  
→ *Perfekt parallelisierbar*

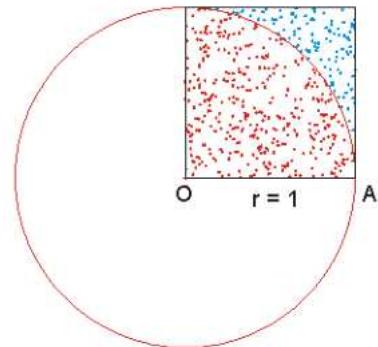
# Einfaches Beispiel

- Bestimmung von  $\pi$

- Fläche des Einheitskreises:

$$F = \pi r^2 = \pi$$

- Bestimme möglichst viele zufällige Punkte im Einheitsquadrat
  - Zähle, wie viele Punkte davon im Einheits(viertel)kreis liegen
  - Aus dem Anteil lässt sich ein Wert für  $\pi$  nähern



# Wiederholung: Threads in Java

---

- Instanz der Klasse **Thread** oder einer Unterklasse davon, die mit der Methode **start()** als Hardware-Thread aktiviert wird (erst dann „lebt“ der Thread).
- 2 Methoden zur Erzeugung von Threads:
  - *Direkte Ableitung von Thread*
  - *Implementation von Runnable*

# Ableitung von **Thread**

---

```
class MyThread extends Thread {  
    ...  
    public void run() {  
        // whatever this thread does  
    }  
    ...  
}  
  
// Thread-Objekt erzeugen  
MyThread t = new MyThread(...);  
  
// Thread aktivieren  
t.start();
```

# Implementation von **Runnable**

---

```
class MyClass implements Runnable {  
    ...  
    public void run() {  
        // whatever this thread does  
    }  
    ...  
}  
  
// Thread-Objekt erzeugen  
Thread t = new Thread(new MyClass(...));  
  
// Thread aktivieren  
t.start();
```

# Implementation von **Runnable**

---

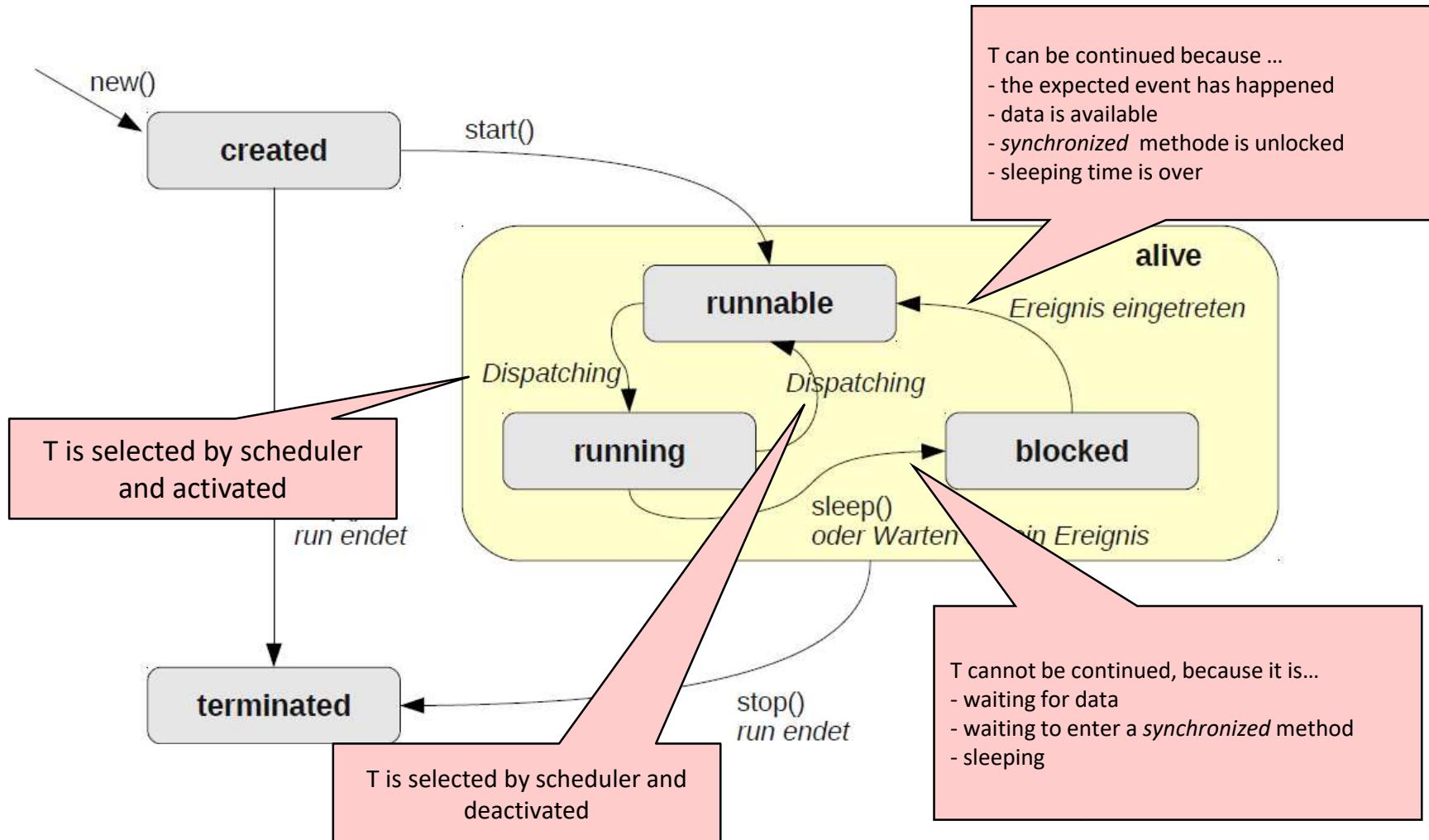
```
class MyClass extends OtherClass implements Runnable {  
    ...  
    public void run() {  
        // whatever this thread does  
    }  
    ...  
}  
  
// Thread-Objekt erzeugen  
Thread t = new Thread(new MyClass(...));  
  
// Thread aktivieren  
t.start();
```

# Gleichzeitigkeit

---

- Threads laufen **logisch** gleichzeitig
  - In der Realität: ???
  - Scheduler (JVM) wechselt Zustände der Threads
    - entscheidet, welcher Thread laufen darf („running“) und welcher warten muss („Runnable“ / „suspended“) → Konkurrenz!
    - muss nicht immer „fair“ sein
    - „willkürliches“ Hin- und Herschalten (switching)
    - (fast) kein Einfluss durch Programmierung
- Programmiere so, als würden alle laufbereiten Threads tatsächlich gleichzeitig ausgeführt (und überlasse den Rest dem System)

# Java Threads: Life cycle



# Thread Scheduling

---

- Threads laufen **logisch** gleichzeitig
- Scheduler (JVM) **wechselt Zustände** der Threads
  - entscheidet, welcher Thread laufen darf („running“) und welcher warten muss („runnable“ / „suspended“) → Konkurrenz!
  - muss nicht immer „fair“ sein
  - „willkürliches“ Hin- und Herschalten (switching)
  - (fast) kein Einfluss durch Programmierung
- Threads werden **verschränkt** oder physikalisch **parallel** (oder beides) ausgeführt

# Was bedeutet Thread Scheduling für das Programm?

---

- Ohne Threads: determiniert
- Mit Threads:
  - Übergänge: *ready*  $\leftrightarrow$  *running* unbekannt!
- Wenn wir so programmieren, als ob alle Aufgaben, die gleichzeitig ausgeführt werden können, tatsächlich gleichzeitig ausgeführt werden, kann dieser Nichtdeterminismus keinen Schaden anrichten.
- Was ist, wenn die Threads nicht völlig unabhängig sind?

# Multi-Threading

---

- Die Aktivitäten der Threads sind oft unabhängig voneinander  
→ Existenz von Threads und Scheduling kann ignoriert werden
- ABER: Manchmal kommen Threads in Konflikt:
  - *Programmteile, die von einem Thread problemlos genutzt werden können, führen zu Problemen, wenn mehrere Threads sie nutzen*

# Was kann passieren?

---

- **Race condition** (Wettrennen)
  - *Mehrere Threads verändern dasselbe Objekt; je nach Scheduling variiert das Resultat*
- **Deadlock** (Verklemmung)
  - *mehrere Threads blockieren sich gegenseitig, weil sie auf Ressourcen warten, die ein jeweils anderer gerade hält*
- **Thread starvation** (Verhungern)
  - *Ein Thread wartet auf eine Ressource, die ihm niemals zugeteilt wird*

# Race Condition – Beispiel

- Unser Array-Zähler (s. Vorlesung 1)

1	0	1	1	1	1	0	0	1	1	0	1	0	1	1	1	1	1	0	1	1	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- Bisher:
  - *Main-Thread wartet auf das Ende aller Threads, holt dann die Ergebnisse ab und summiert sie*
  - *Problem: bereits beendete Threads müssen warten*
- Alternative Idee:
  - *Jeder Thread zählt, sobald er seine Summe berechnet hat, sein Ergebnis zum Gesamtergebnis hinzu*
  - *Sobald der letzte Thread beendet ist, hat man sofort das Ergebnis*

# Vorher

```
class CountThread extends Thread {  
    private boolean[] array;  
    private int lo;  
    private int hi;  
    private int result;  
  
    public CountThread(boolean[] array, int lo, int hi) {  
        this.array = array;  
        this.lo = lo;  
        this.hi = hi;  
    }  
  
    public void run() {  
        for(int i = lo; i <= hi; i++)  
            if(array[i])  
                result++;  
    }  
  
    public int getResult() {  
        return result;  
    }  
}
```

Übergib Parameter  
im Konstruktor!

Übergib Ergebnis  
mittels Methode!

# Vorher

```
// Synchronisation (auf alle Threads warten)

try {
    for(int i = 0; i < NUMBER_OF_THREADS; i++) {
        counter[i].join();
    }
} catch(InterruptedException e) {
    e.printStackTrace();
}

// Gesamtergebnis aus Teilergebnissen berechnen
int result = 0;
for(int i = 0; i < NUMBER_OF_THREADS; i++) {
    result += counter[i].getResult();
}
```

`join()` führt den Thread, für den es aufgerufen wird, mit dem aufrufenden Thread zusammen.

# Jetzt

```
class CountThread extends Thread {  
    private boolean[] array;  
    private int lo;  
    private int hi;  
    private int result;  
    static int overall_result = 0; // Gemeinsame Variable für Gesamtergebnis  
  
    public CountThread(boolean[] array, int lo, int hi) {  
        this.array = array;  
        this.lo = lo;  
        this.hi = hi;  
    }  
  
    public void run() {  
        for(int i = lo; i <= hi; i++)  
            if(array[i])  
                result++;  
        overall_result += result;  
    }  
}
```

Zähle Ergebnis zum  
Gesamtergebnis hinzu.

# Array-Count: Ergebnis

---

```
// Synchronisation (auf alle Threads warten)

try {
    for(int i = 0; i < NUMBER_OF_THREADS; i++) {
        counter[i].join();
    }
} catch(InterruptedException e) {
    e.printStackTrace();
}
System.out.println("Result = " + CountThread.overall_result);
```

Funktioniert das?

# Race conditions

---

- Unterschiedliche Ausführungen führen zu unterschiedlichen und inkorrekten Ergebnissen
- können Klassen Thread-**unsicher** machen
- sind in der Regel zu vermeiden (es gibt Ausnahmen)

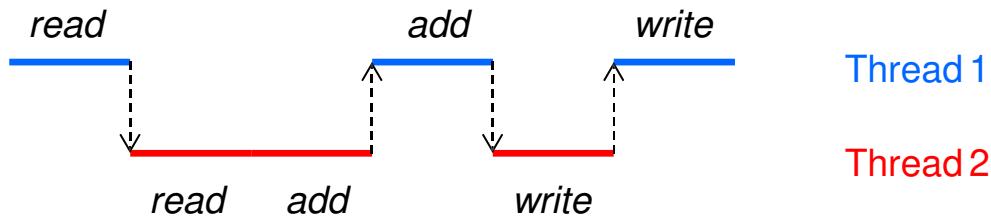
# Zugriff auf gemeinsame Ressourcen

---

- Synchronisation notwendig, um Race Conditions zu vermeiden (Teilergebnisse werden „vergessen“)
- Die zu verändernde Ressource muss vor simultanen Änderungen geschützt werden

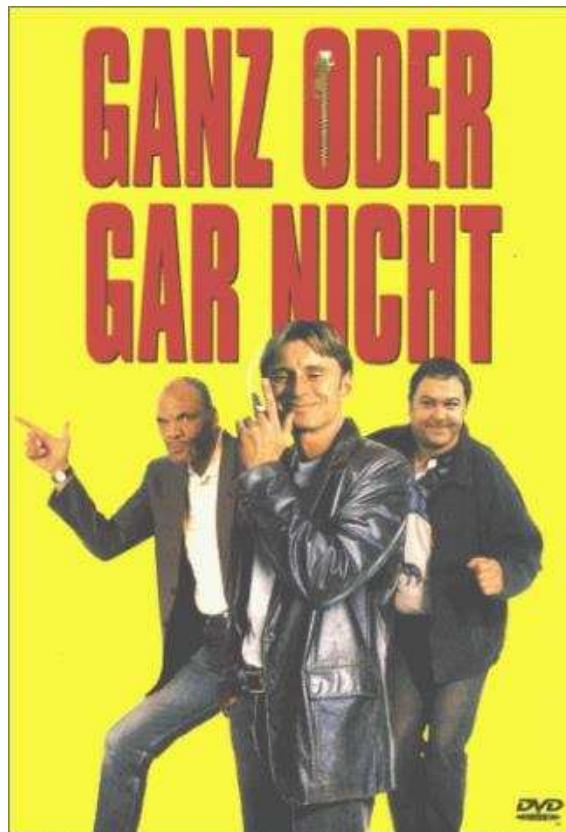
# Problem

- Aufaddieren ist keine **atomare** Operation  
**overall\_result += result** entspricht:
  - Lies **overall\_result** in ein Register **x**
  - Addiere den Wert von **result** zu **x**
  - Schreibe das Ergebnis zurück in **overall\_result**
- Zwischen den Teiloperationen kann ein Kontextwechsel (Thread-Switching) stattfinden



# Atomizität

- Eine Operation ist **atomar**, wenn sie von einem Thread nur entweder



ausgeführt wird.

# Atomizität in Java

- Einfaches Dazu-Addieren ist nicht threadsicher  
(weil es nicht **atomar** ist)
- **Atomar** sind für „kleine“ primitive Datentypen  
(**int, char, byte, short, float, boolean**):
  - *Einfaches Lesen*
  - *Einfaches Schreiben*
- aber **nicht** für **long, double**
- Für allgemeine Objekte (Referenztypen) gibt es keine garantiert atomaren Operationen
  - Verantwortung bei Programmierer

Nicht beides zusammen!

# Thread-Sicherheit

---

- Eine Klasse ist **threadsicher**, wenn sie auch dann **korrekt** bleibt, wenn ihr Code von **mehreren Threads** ausgeführt wird, unabhängig von
  - *Scheduling der Threads*
  - *sonstigen Verschränkungen*
  - *Code ausführen* betrifft alle öffentlich zugänglichen
    - statischen Felder und Methoden
    - Methoden und Felder beliebiger Instanzen
  - *Korrekt* bedeutet: *Verhalten gemäß Spezifikation*

# Wie kann man Thread-Sicherheit herstellen?

---

- Operationen **atomar** machen
  - *durch „Aussperren“ anderer Threads während der Ausführung*
- **Zustandslosigkeit** von Klassen
  - *Keine gemeinsam verwendeten Ressourcen*
  - *Jede Methode stellt ihre eigenen Variablen/Objekte bereit*

# Zustandslosigkeit

---

- Garantiere, dass jeder Thread seine eigenen Instanzen von allen verwendeten Feldern bekommt
  - Klasse bzw. Objekt hat keinen Zustand, der sich ändern kann (*Immutability*)
  - z.B. bei Klassen, die als Service fungieren und eine Berechnung vornehmen, die nur auf dem Input der Threads beruht
  - in der Praxis
    - oft nicht möglich
    - noch öfter nicht sinnvoll

# Synchronisation

---

- Garantiere, ...
  - dass zu jedem Zeitpunkt nur ein Thread den Zustand eines Objekts (bzw. einer Klasse) verändern kann (*mutual exclusion*)  
*und*
  - dass beim Auslesen immer ein sinnvoller, d.h. konsistenter Zustand (gemäß Spezifikation) herrscht

# Synchronisation

---

- Eliminiert den Nichtdeterminismus aufgrund von Nebenläufigkeit
- Unerwünschte Ablaufsequenzen ausschließen
- Zwei Arten:
  - *Wettbewerbssynchronisation*
    - Behandlung von **Konkurrenz**
    - Mittel: gegenseitiger Ausschluss (mutual exclusion)
  - *Bedingungssynchronisation*
    - Ermöglichen von **Kooperation**
    - Mittel: Bedingungen, Barrieren

# Kooperatives Multitasking

---

- Ziel:
  - *nicht, einen Thread vor einem anderen zu schützen, sondern:*
  - *Gesamtprogramm läuft sinnvoll / korrekt / schneller*
- Threads machen Programme nicht-deterministisch:
  - *Nicht Programm entscheidet allein über den exakten Ablauf, der Scheduler (Teil der JVM) entscheidet mit darüber*
  - *Nichtdeterminismus ist meist erwünscht*
  - *Gelegentlich muss der Nichtdeterminismus wieder reduziert werden:*
    - In Synchronisationssituationen muss der Willkür des Schedulers Grenzen gesetzt werden:  
→Programm beeinflusst (synchronisiert) die Threadausführung!

# Kooperatives Multitasking

Total nicht-deterministische Programmausführung (keine Synchronisation)



**Threading**, eingeschränkter Nicht-Determinismus:  
Atomare Aktionen werden nicht unterbrochen  
alles innerhalb eines Threads wird sequenziell ausgeführt

**Synchronisation**skonstrukte:  
Zusätzliche Beschränkung des Nicht-Determinismus  
(nur so viel wie nötig, so wenig wie möglich)

Total deterministische Programmausführung (komplett sequenziell)

# Wettbewerbssynchronisation

---

- Atomizität garantieren
- Möglichkeiten:
  - *Atomare (durch Java-Spezifikation) Konstrukte verwenden*
    - Bibliotheken, z.B. `java.util.concurrent.atomic`
  - *Atomizität erzwingen durch Sperren (Locks, Monitor)*
    - Explizit programmiert: `synchronized`
    - Implizit durch Bibliotheken, z.B. `java.util.concurrent.atomic`

# Explizite Sperren

- In Java mit dem Keyword **synchronized**
- Verwendbar für
  - Methoden:  
`synchronized void setValue(... x) {  
 this.x = x;  
}`
  - Code-Blöcke  
`void setValue(... x) {  
 ...  
 synchronized(o) {  
 this.x = x;  
 }  
}`
- Es wird jeweils eine **Sperre** (**Monitor** bzw. **Lock**) gesetzt

# synchronized

---

- Bei **nicht-statischen** Methoden wird der **Monitor** des **Objekts** genutzt, dessen **synchronized**-Methode aufgerufen wird
- Bei **statischen** Methoden wird der **Monitor** der **Klasse** genutzt, in der die **synchronized**-Methode definiert ist
- Bei **Code-Blöcken** muss ein (**beliebig wählbares**) **Objekt** angegeben werden, dessen **Monitor** verwendet wird

# Monitor

- Jedes Java-Objekt hat einen Monitor
  - entweder *gesperrt* (wird von einem Thread „gehalten“)
  - oder *frei*
- Beim Aufruf einer synchronized-Methode/Block
  - Wenn Monitor *frei*, dann
    - Monitor wird gesperrt (von diesem Thread gehalten)
    - Methode/Block betreten und ausgeführt
  - Wenn Monitor *gesperrt*, dann
    - Thread wird vom Scheduler in **blockiert**-Zustand gesetzt (Ausnahme: derselbe Thread hat den Monitor schon)



# Monitor

- Jedes Objekt hat einen Monitor
  - entweder **gesperrt** (wird von einem Thread „gehalten“)
  - oder **frei**
- Beim Verlassen einer synchronized-Methode/Block
  - Alle wegen dieses Monitors **blockierten** Threads gehen in Zustand **ready**
  - Einer davon bekommt den Monitor, **sperrt** ihn und betritt die Methode / den Block
  - Alle anderen werden sofort wieder **blockiert**



# Eigenschaften von Monitoren/Locks

```
public class C {  
    private Object o = new Object();  
  
    public synchronized void m1() {  
        ...  
    }  
  
    public void m2() {  
        synchronized (this) {  
            ...  
        }  
        synchronized (o) {  
            ...  
        }  
    }  
  
    public synchronized void m3() {  
        synchronized (this) {  
            ...  
        }  
        synchronized (o) {  
            ...  
        }  
    }  
}
```

**synchronized** bezieht  
sich auf Lock-Objekte

Vom gleichen Lock geschützt  
(Ein -Lock je Instanz der Klasse C)

Vom gleichen Lock geschützt  
(Ein -Lock je (Wert von) o)

doppelt gelockt

Welche Aufrufe / welche nebenläufigen  
Aktivitäten sind möglich ?

# Eigenschaften von Monitoren/Locks

```
public class C {  
    private Object o = new Object();  
  
    public synchronized void m1() {  
        ...  
        m1();  
        m2();  
        ...  
    }  
  
    public synchronized void m2() {  
        ...  
        m3();  
    }  
  
    public void m3() {  
        synchronized (o) {  
            ...  
        }  
    }  
}
```

## Java-Locks sind wiederbetretend (*reentrant*)

Ein Lock-Objekt kann von einem Thread beliebig oft belegt werden.  
(Wer schon drin ist, darf nochmal rein!)

Anderes Lock-Objekt,  
vielleicht von einem anderen  
Thread belegt

# Zusätzliche Wartebedingungen

---

Manchmal ist es nötig, dass Threads warten, obwohl kein Monitor belegt ist, weil das Objekt aus anderen Gründen nicht bearbeitet werden kann.

Typisches Beispiel: Producer-Consumer-Pattern

- Datenstruktur mit begrenzter Kapazität (s. Labor 2)
  - Einfügen erst, wenn wieder Platz ist
  - Herausnehmen nur, wenn etwas zum Entfernen da ist

Gleichzeitig möchte man möglichst verhindern, dass das Warten Rechnerressourcen kostet (*busy waiting*)

# Wartende Threads

---

## Möglichkeiten

- (1) Thread wartet selbst „außerhalb“ des Objekts, z.B.

```
while (stack.isFull()) {  
    ...  
}  
stack.push(...);
```

### Probleme:

*Busy waiting*, kostet Rechnerressourcen (Abhilfe: *sleep*)

Thread kommt möglicherweise nie zum Zug

Kontextwechsel nach **while** und vor **push** → *Race condition*

- (2) Warten innerhalb des Objekts in der **synchronized**-Methode

# Warten innerhalb der Methode

---

1. Versuch:

```
synchronized void push(long k) {  
    while (isFull()) {  
        ...  
    }  
    h[end++] = k;  
}
```

Noch schlechter:

Methode wird nie mehr verlassen, wenn `isFull() == true`

# Warten innerhalb der Methode

---

2. Versuch: push nicht mehr synchronized

```
void push(long k) {  
    while (isFull()) {}    // isFull synchronized  
    h[end++] = k;  
}
```

Auch nicht korrekt:

Methoden-Inhalt **nicht atomar!**

Kontextwechsel zwischen Anweisungen möglich.

# Lösung: wait / notify

---

Methoden von `java.lang.Object`

können nur innerhalb von `synchronized` Methoden oder Blöcken aufgerufen werden

`wait()` kann eine `InterruptedException` werfen

`wait()`

veranlasst den aktuellen Thread zu warten (der Scheduler blockiert ihn  
→ kein busy waiting!)

Gleichzeitig wird der `Monitor` des Objekts **freigegeben** (sonst könnte kein anderer Thread etwas im Objekt tun, das ebenfalls `synchronized` ist)

# Warten mit `wait()`

---

Korrekt:

```
synchronized void push(long k) {  
    while (isFull()) {  
        try {  
            wait();  
        } catch(InterruptedException e) { ... }  
    }  
    h[end++] = k;  
}
```

# wait / notify

---

## notify()

- veranlasst den Scheduler, **einen** der Threads zu benachrichtigen, der wegen dieses Objekts wartet
- Der benachrichtigte Thread muss aber den **Monitor** erst wieder akquirieren, bevor er weitermachen kann!
- erst am **Ende** der Methode, aus der `notify` aufgerufen wurde, wird das Lock vom aktuellen Thread freigegeben

## notifyAll()

- Wie `notify`, aber **alle** Threads, die wegen des Objekts warten, werden benachrichtigt
- Kann immer statt `notify` verwendet werden, aber nicht umgekehrt!
- weniger effizient, wenn viele Threads existieren

# Benachrichtigung mit `notify()`

---

Notify:

```
synchronized long pop() {  
    long result = h[--end];  
    notifyAll();  
    return result;  
}
```

# Kombination wait/notify

---

Komplett:

```
synchronized void push(long k) {  
    while (isFull()) {  
        try {  
            wait();  
        } catch(InterruptedException e) { ... }  
    }  
    h[end++] = k;  
    notifyAll();  
}
```

# Kombination wait/notify

---

Notify:

```
synchronized long pop() {  
    while (isEmpty()) {  
        try {  
            wait();  
        } catch(InterruptedException e) { ... }  
    }  
    notifyAll();  
    return h[--end];  
}
```

# Threadsichere Datenstrukturen

---

Threadsichere Versionen diverser Datenstrukturen sind verfügbar im Package **java.util.concurrent**

Beispiele blockierender Datenstrukturen:

ArrayBlockingQueue

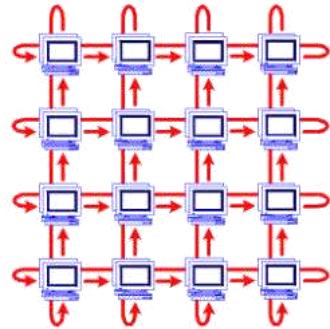
LinkedBlockingQueue

SynchronousQueue

Nicht-blockierende, aber threadsichere Datenstrukturen

ConcurrentLinkedQueue

...



---

# Parallel Computing

Tobias Lauer

Hochschule Offenburg

---

# Topics

---

- Grenzen der Parallelisierung
- Das PRAM Modell
- Analyse paralleler Algorithmen

# Grenzen der Beschleunigung

---

- Wie sehr kann man eine Anwendung überhaupt durch Parallelisierung beschleunigen?
- Idealisierte Annahmen
  - *wir haben so viele Prozessoren, wie wir wollen*
  - *unser parallelisierter Anteil ist mit  $p$  Prozessoren  $p$ -mal so schnell wie mit einem*
- Was ist die Obergrenze für den Speedup?
  - Zur Erinnerung: Speedup  $S(p) = t_{seq} / t_{par(p)}$

# Amdahlsches Gesetz

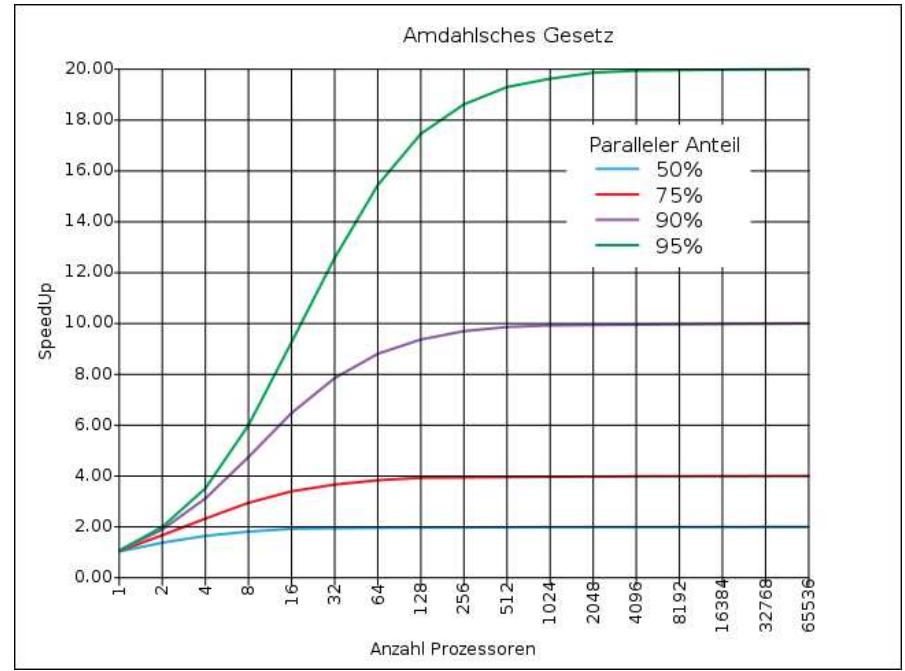
(Gene Amdahl, 1922-2015)

---

- Jede Anwendung hat auch einen nicht parallelisierbaren Anteil, z.B. Laden, Initialisieren von Variablen, etc.
- Sei  $0 < \text{Par} < 1$  der parallelisierbare Anteil der Laufzeit  $t_{seq}$
- Wenn 1 die Gesamtlaufzeit ist, dann beträgt die Zeit für den nicht parallelisierbaren Teil  $(1 - \text{Par})$ , da  $1 = \text{Par} + (1 - \text{Par})$
- Beschleunigen lässt sich nur Par: bei  $p$  Prozessoren lässt sich die Zeit höchstens bis auf  $\text{Par} / p$  reduzieren.
- Die Gesamtzeit wäre dann  $(1 - \text{Par}) + \text{Par} / p$
- D.h. schneller als  $(1 - \text{Par})$  können wir niemals werden, egal wieviele Prozessoren wir haben!
- Speedup  $S(p) = 1 / (1 - \text{Par} + \text{Par}/p) \leq 1 / (1 - \text{Par})$

# Amdahl's Law: Beispiel

- Der nicht-parallele Teil eines Programms nimmt 10% seiner Laufzeit in Anspruch, 90% sind (perfekt) parallelisierbar ( $\text{Par} = 0.9$ ;  $1-\text{Par} = 0.1$ )
  - Mit 2 Prozessoren brauchen wir*
    - $0.45 + 0.1 = 0.55$  der Zeit  
Speedup: 1.8x
  - Bei 4 Prozessoren sind es*
    - $0.225 + 0.1 = 0.325$   
Speedup: 3.1x
  - Bei 16 Prozessoren*
    - $0.056 + 0.1 = 0.156$   
Speedup: 6.4x
  - Bei 64 Prozessoren*
    - $0.014 + 0.1 = 0.114$   
Speedup: 8.8x



Quelle: Wikipedia

# Amdahl = schlechte Nachricht?

---

- Ja, für Probleme **fester** Größe
  - *Diese lassen sich mit immer mehr Prozessoren nur bis zu einem bestimmten Punkt beschleunigen*
- Amdahl's Law nimmt an, dass der parallelisierbare Anteil **Par** von der Anzahl **p** der Prozessoren unabhängig ist
- In der Realität sieht es oft anders aus:
  - *Man möchte in derselben Zeit ein größeres Problem lösen und verwendet dafür mehr Prozessoren*
  - *In diesem Fall wird in der Regel auch Par deutlich größer, weil der nicht parallelisierbare Programmteil häufig nahezu konstant ist*

# Gustafson's Law

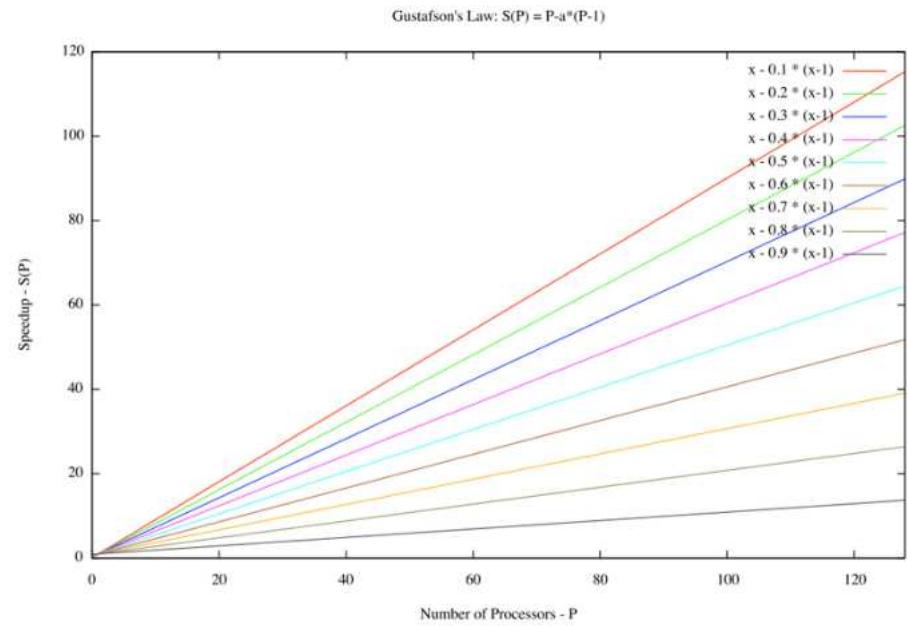
(John Gustafson, 1988)

---

- Andere Perspektive auf **Beschleunigung (Speedup)**:  
Mit mehr **Prozessoren** darf sich die **Problemgröße** erhöhen
- Gustafson:  $S(p) = (1 - \text{Par}) + p \cdot \text{Par}$   
„Wieviel **mehr** schaffe ich in **derselben Zeit** mit  **$p$**  Prozessoren statt mit **einem?**“
- Vgl. Amdahl:  $S(p) = 1 / (1 - \text{Par} + \text{Par}/p)$   
„Um wieviel **kürzer** löse ich **dasselbe Problem** mit  **$p$**  Prozessoren als mit **einem?**“

# Gustafson's Law: Beispiel

- Der nicht-parallele Teil eines Programms nimmt 10% seiner Arbeit in Anspruch, 90% sind (perfekt) parallelisierbar ( $\text{Par} = 0.9$ ;  $1-\text{Par} = 0.1$ )
  - Mit 2 Prozessoren schaffen wir das
    - $0.1 + 2 * 0.9 = 1.9$ -fache Speedup: 1.9x
  - Bei 4 Prozessoren ist es das
    - $0.1 + 4 * 0.9 = 3.7$ -fache Speedup: 3.7x
  - Bei 16 Prozessoren das
    - $0.1 + 16 * 0.9 = 14.5$ -fache Speedup: 14.5x
  - Bei 64 Prozessoren das
    - $0.1 + 64 * 0.9 = 57.7$ -fache Speedup: 57.7x

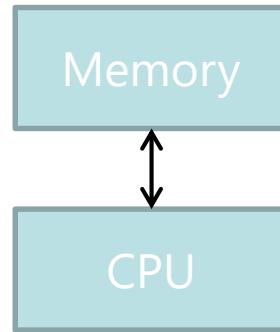


# Parallele Rechenmodelle

- Sequenzielles Rechenmodell

- *Von-Neumann-Architektur:*

*Ein Programm (Machinencode) wird im Speicher angelegt und ausgeführt, indem eine Instruktion nach der anderen an die CPU übergeben wird.*



- *Es gibt nur einen Speicher für Daten und Befehle*
  - *Relative enge Beziehung zwischen Hardware, Software und theoretischen Modellen*

# Parallele Rechenmodelle

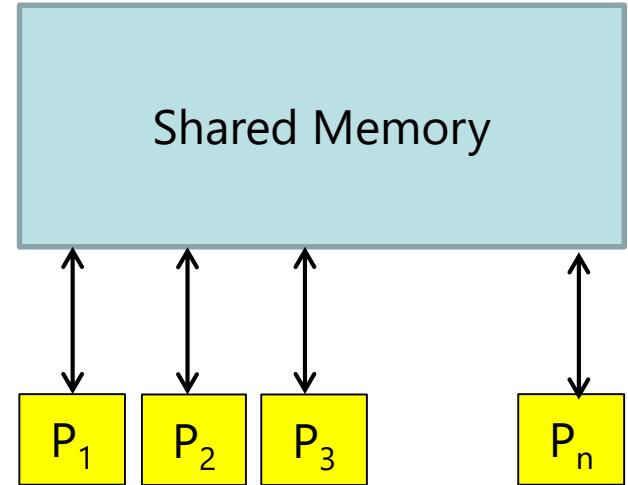
---

- Diverse parallele Rechen- und Rechnermodelle
- Jedes wird anders programmiert
- Daher hat man beim Entwurf von Algorithmen stets ein konkretes Modell im Auge (z.B. Editierdistanz)
- Verschiedene parallele Maschinen eignen sich für die Lösung unterschiedlicher Probleme

# Das PRAM Modell

- Parallel Random Access Machine

- *n Prozessoren, verbunden mit einem gemeinsamen Speicher*
- *Jeder Prozessor kann in jedem Taktzyklus auf einen beliebigen Ort im Speicher zugreifen*
- *Es kann zu Konflikten beim Zugriff kommen. Das Memory Management der Hardware muss das regeln.*



# PRAM

---

- Theoretisches Modell
- Es gibt keine Hardware, die es genau so umsetzt
- ignoriert viele konkrete Dinge
  - z.B. Cache, ...
- trotzdem hilfreich
  - *Nicht für jede konkrete Architektur eigener Algorithmus nötig*
  - *An diverse Architekturen mit leichten Änderungen anpassbar*

# Parallele Architekturen

- Klassifikation paralleler Architekturen (nach Flynn, 1966):

	Single instruction	Multiple instruction
Single Data	SISD	MISD*
Multiple Data	SIMD	MIMD

- Beispiele: ?

# MIMD

---

- Die Prozessoren verarbeiten verschiedene Programme/Algorithmen, die von Zeit zu Zeit kommunizieren
- Geeignet zur Lösung von Problemen ohne besondere innere Struktur
  - *Verschiedenartige Tasks*
  - *Austausch/Kommunikation in unregelmäßigen Abständen*

# SIMD

---

- Auf jedem Prozessor läuft derselbe Algorithmus, aber mit jeweils unterschiedlichem Input
- Geeignet zur Lösung von Problemen mit regulärer interner Struktur
  - *Gleichartige Tasks auf einer (aufteilbaren) Menge von Daten*
  - *Kommunikation/Synchronisation in regelmäßigen Abständen*
  - *Es gibt viele solcher Probleme*  
*(z.B. Grafikrendering, Array-Count, Editierdistanz, ...)*

# SIMD Eigenschaften

---

- Ein  $n$ -Prozessor SIMD-Computer hat die folgenden Eigenschaften:
  - Jeder Prozessor kann sowohl *Instruktionen* als auch *Daten* in seinem lokalen Speicher (Register) ablegen.
  - Jeder Prozessor hat eine *identische Kopie desselben Programms* in seinem Speicher.
  - In jedem Taktzyklus führt jeder Prozessor *dieselbe Instruktion* dieses Programms aus. Jedoch *unterscheiden* sich die *Daten* (Input) der verschiedenen Prozessoren.
  - Die Prozessoren kommunizieren entweder über ein *Netzwerk* oder durch einen *gemeinsamen Speicher*.

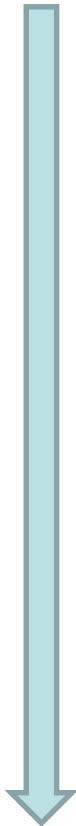
# SIMD mit Shared Memory

---

- Klassifikation der Zugriffs (PRAM):
  - *Exclusive Read Exclusive Write (EREW)*  
*Auf jeden Speicherort kann nur von **einem** Prozessor gleichzeitig zugegriffen werden.*
  - *Concurrent Read Exclusive Write (CREW)*  
*Mehrere Prozessoren können gleichzeitig aus demselben Speicherort lesen, aber nur **ein** Prozessor kann zu einer Zeit schreiben.*
  - *Concurrent Read Concurrent Write (CRCW)*  
*Mehrere Prozessoren können gleichzeitig lesend oder schreibend auf denselben Speicherort zugreifen.*
    - Wie wird gleichzeitiges Schreiben geregelt?

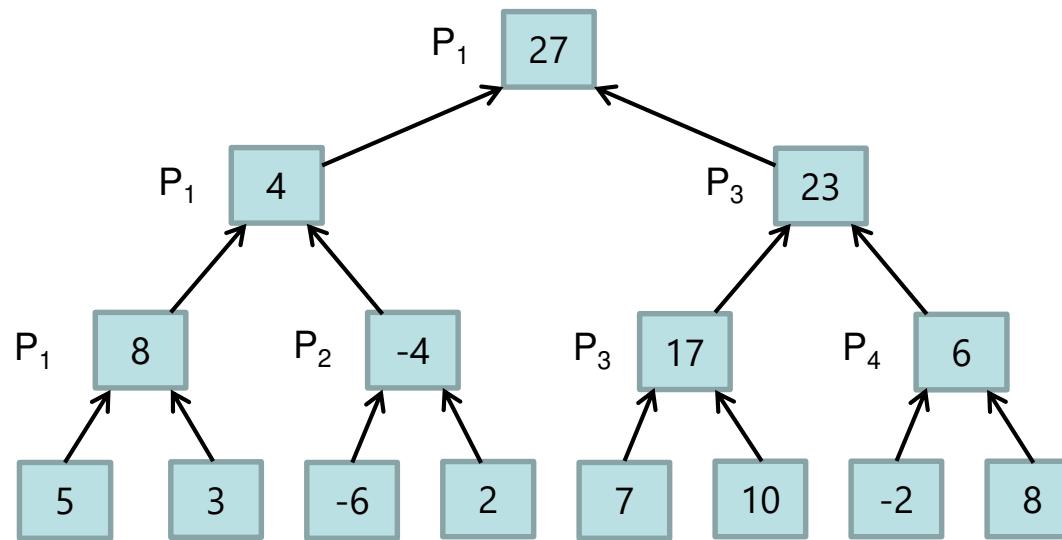
# CRCW Varianten

- Common CRCW
  - Alle Prozessoren, die schreiben wollen, müssen denselben Wert schreiben
- Arbitrary CRCW
  - Einer der Prozessoren „gewinnt“ (sein Wert steht am Ende am Speicherort). Welcher, wissen wir nicht!
- Priority CRCW
  - Prozessoren werden mit Prioritäten versehen; der mit der höchsten Priorität schreibt seinen Wert



# Parallele Algorithmen

- Einfaches Beispiel:  $n$  Zahlen parallel addieren
- Anders als bisher:



Zeit:  $O(\log n)$

Prozessoren:  $n/2 = O(n)$

# Analyse paralleler Algorithmen

---

- Zeitkomplexität  $T_p(n)$  = #Zeitschritte mit  $p$  Prozessoren
- Prozessorkomplexität  $P(n)$  = #Prozessoren (in Abhängigkeit von  $n$ )
- Work  $W(n)$  = Anzahl primitiver Operationen, die auf allen Prozessoren zusammen ausgeführt werden.  
→ entspricht  $T_1(n)$ , also der sequentiellen Ausführung
- Span (auch: *depth*, *critical path length*): Anzahl primitiver Operationen auf dem längsten sequentiellen Pfad  
→ entspricht  $T_\infty(n)$ , also der Ausführung auf einer idealisierten Maschine mit unendlich vielen Prozessoren
- Cost:  $C_p(n) = p \cdot T_p(n)$  = Kosten der Ausführung

# Analyse paralleler Algorithmen

---

- Unser Beispiel:

- $T_p(n) =$

- $P(n) =$

- $W(n) =$

- $T_\infty(n) =$

- $C_p(n) =$

# Eigenschaften

---

- Work law:  $p \cdot T_p \geq T_1$ 
  - Die **Cost** entspricht immer mind. der **Work**. (Denn  $p$  Prozessoren können höchstens  $p$  Operationen parallel ausführen).
- Span law:  $T_p \geq T_\infty$ 
  - Mit endlich vielen Prozessoren kann man nicht schneller sein also mit unendlich vielen (aber womöglich gleich schnell!)

# Eigenschaften

---

- Speedup  $S_p = T_1 / T_p$
- Aus dem *Work law* folgt:  $T_1 / T_p \leq p$   
→  $S_p$  ist beschränkt durch  $p$
- $S_p / p$  wird auch als Efficiency bezeichnet
- Parallelism  $T_1 / T_\infty$  ist der höchstmögliche Speedup ( $S_\infty$ )

# Eigenschaften

- Slackness: 
$$T_1 / pT_\infty \leq T_1 / pT_p \leq T_1 / T_1 = 1$$

↑                              ↑  
span law                    work law
- Wenn Slackness < 1 → Speedup  $S_p < p$ 
  - Keine perfekte Parallelisierbarkeit

# Optimalität bei parallelen Algorithmen

---

- Bei sequenziellen Algorithmen:
  - *Bestimme untere asymptotische Schranke  $T(n)$  für ein Problem  $P$  der Größe  $n$* 
    - Beispiel:  $T(n) = \Omega(n \log n)$  für das Sortieren beliebiger  $n$  Zahlen
  - *Falls Algorithmus A das Problem für alle  $n$  in Zeit  $T(n)$  löst, ist A optimal für P*
    - Beispiel: Merge-Sort, Heapsort, einige Varianten von Quicksort
- Betrachte einen parallelen Algorithmus, der  $P$  der Größe  $n$  mit Work  $W(n)$  in Zeit  $T_p(n)$  löst.

# Optimalität bei parallelen Algorithmen

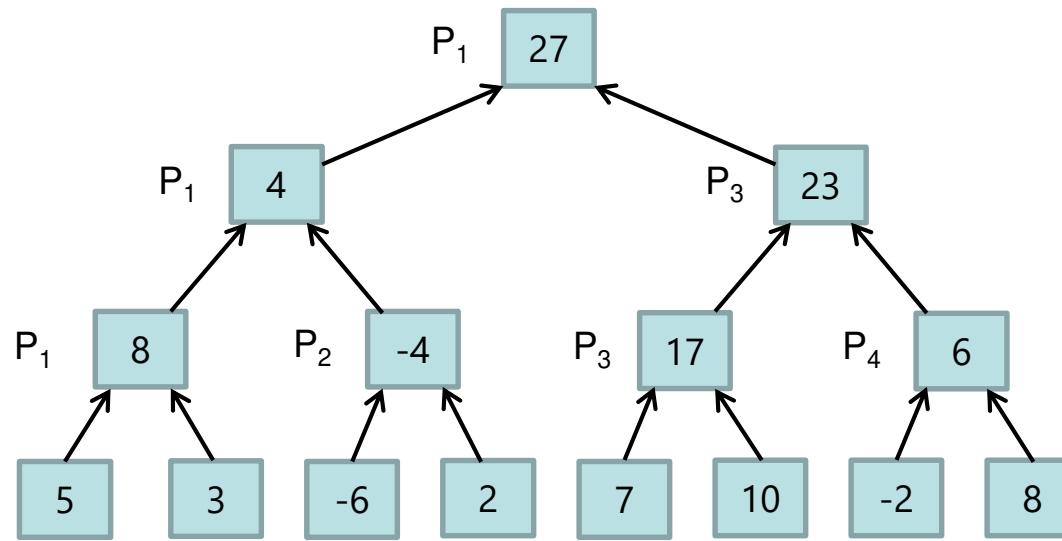
---

Sei  $T_{\text{seq}}(n)$  die Laufzeit des besten sequentiellen Algorithmus

- Ein paralleler Algorithmus heißt **work-optimal**, falls
  - $W(n) = T_1(n) = O(T_{\text{seq}}(n))$
- Er heißt außerdem **work-time-optimal**, falls
  - $T_p(n)$  nicht weiter verbessert werden kann
- Ein paralleler Algorithmus heißt **cost-optimal**, falls
  - $c(n) = p \cdot T_p(n) = O(T_{\text{seq}}(n))$

# Unser Beispiel

- $n$  Zahlen parallel addieren



# Kennzahlen am Beispiel

- Span:  $T_\infty(n) = O(\log n)$
- Work:  $T_1(n) = O(n)$  ( $n-1$  Additionen)
- Parallelism:  $T_1(n) / T_\infty(n) = O(n) / O(\log n) = O(n / \log n)$

Mit  $p = n/2$  Prozessoren:

- Time:  $T_p(n) = O(\log n)$
- Cost:  $p \cdot T_p(n) = n/2 \cdot O(\log n) = O(n \log n)$
- Speedup:  $S_{n/2} = T_1 / T_{n/2} = O(n / \log n)$
- Efficiency:  $S_p / p = S_{n/2} / (n/2) = O(1 / \log n)$

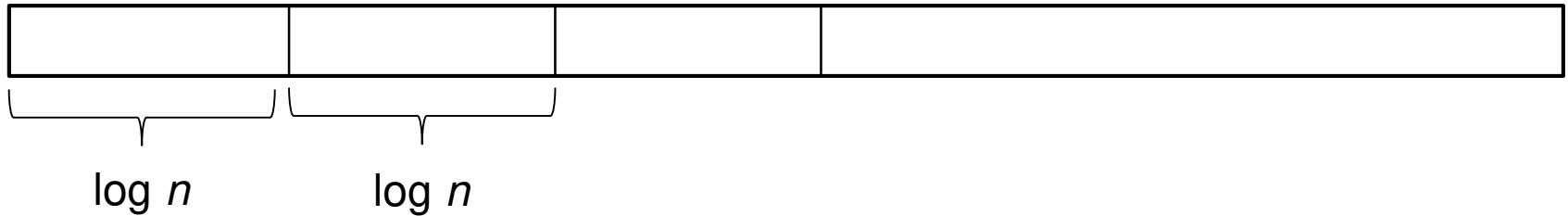
# Unser Beispiel

---

- ... ist **nicht** cost-optimal!
  - $\text{cost}(n) = p \cdot T_p(n) = O(n) \cdot O(\log n) = O(n \log n)$
  - Der sequenzielle Algorithmus benötigt nur Zeit  $O(n)$ .
  - Unser früherer paralleler Algorithmus (Array-Count)  
mit einer **konstanten** Zahl  $p$  von Prozessoren war cost-optimal:
    - $T(n) = n/p + p = O(n) + O(1) = O(n)$
    - $p = O(1)$
  - Auch mit  $n^{1/2}$  Prozessoren war er cost-optimal:
    - $T(n) = n / n^{1/2} + n^{1/2} = O(n^{1/2})$
    - $p = n^{1/2} = O(n^{1/2})$
- Kann man den neuen Algorithmus cost-optimal machen?
  - Ja, man kann **p** reduzieren.

# Reduktion der Prozessorzahl

- Idee – Schritt 1:
  - *Wie bei Array-Count:*
    - Gib jedem Prozessor einen (größeren) Teil des Arrays
    - Jeder Prozessor addiert zunächst seinen Teil **sequenziell** auf
  - *Aber:*
    - Achte darauf, dass dabei die Zeit pro Prozessor in  $O(\log n)$  bleibt



Insgesamt:       $n / \log n$  Teile      →       $n / \log n$  Prozessoren

# Zeit pro Prozessor

- Jeder der  $n / \log n$  Prozessoren summiert zunächst  $\log n$  Zahlen
- Danach sind  $n / \log n$  Zahlen übrig
- Diese können mit den  $n / \log n$  Prozessoren mit Hilfe des Original-Algorithmus in der Zeit

$O(\log(n / \log n))$  addiert werden.

- Gesamtzeit:  $T(n) = O(\log n) + O(\log(n / \log n))$   
 $= O(\log n)$

# Neue Cost

---

- Vorher:
  - $T(n) = O(\log n)$
  - $P(n) = O(n)$
  - $C(n) = O(n \log n)$
- Jetzt:
  - $T(n) = O(\log n)$
  - $P(n) = O(n/\log n)$
  - $C(n) = O(n/\log n) \cdot O(\log n)$   
=  $O(n)$

# Parallele Präfixsumme

---

- Problem:
  - *Gegeben:*
    - Array A von n Zahlen  
 $a_0, a_1, a_2, \dots, a_{n-1}$
    - Binärer assoziativer Operator  $\oplus$  (z.B. +, MAX, MIN)
  - *Gesucht:*
    - Array  
 $a_0, (a_0 \oplus a_1), (a_0 \oplus a_1 \oplus a_2), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-1})$

# Beispiel

---

- $\oplus$  sei die Addition und der Input sei

5, 3, -6, 2, 7, 10, -2, 8

- Dann ist der Output

5, 8, 2, 4, 11, 21, 19, 27

- Rechenzeit sequenziell:

$O(n)$

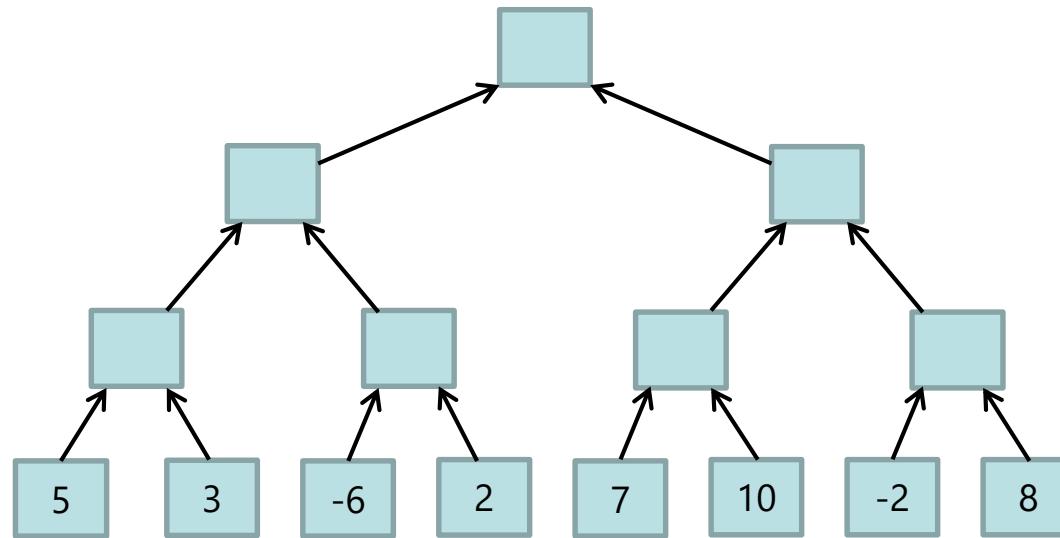
# Wie berechnet man das parallel?

---

- Zunächst: Warum überhaupt?
  - *oft benötigter Baustein für andere parallele Algorithmen*
  - „parallel primitive“

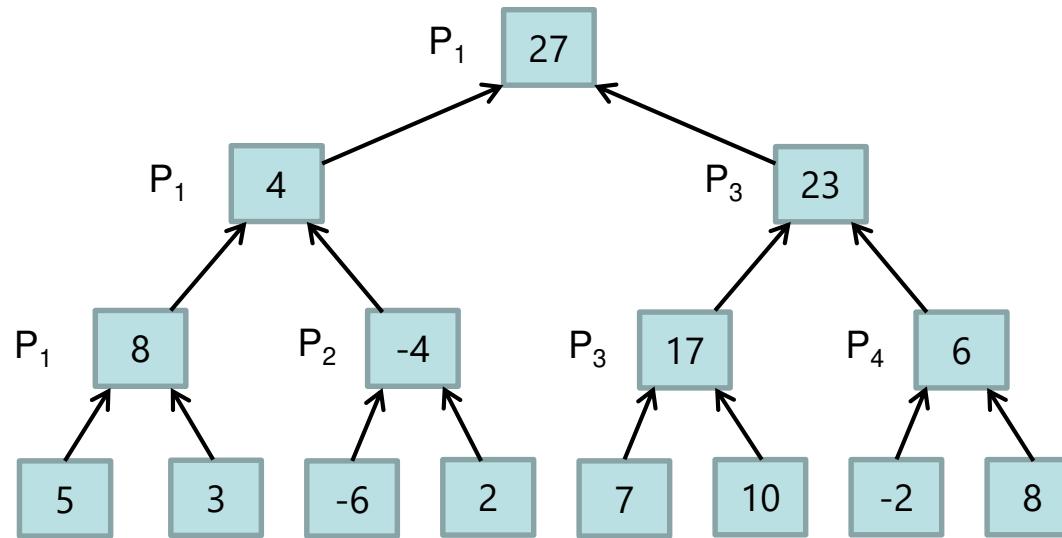
# Parallele Präfixsumme

- Zwei Durchgänge durch Array
  - *Bottom-up: Gesamtsumme berechnen*
  - *Top-down: Teilsummen verteilen*
- Repräsentation als Baum:



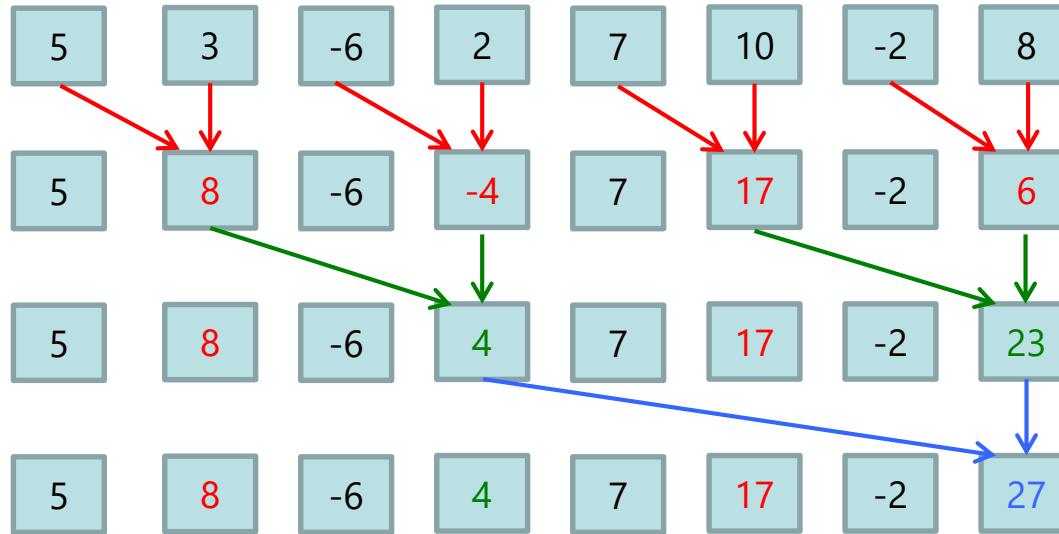
# Parallele Präfixsumme

- Erster Durchgang: **bottom-up**
  - $sum[v] = sum[v.left] + sum[v.right]$
  - Geht auch „in-place“



# Pseudocode

```
for d=0; d<=log(n)-1; d++ do
    for i=0; i<=n-1; i+=2d+1 in parallel
        a[i + 2d+1 - 1] = a[i + 2d - 1] + a[i + 2d+1 - 1]
```



# Parallele Präfixsumme

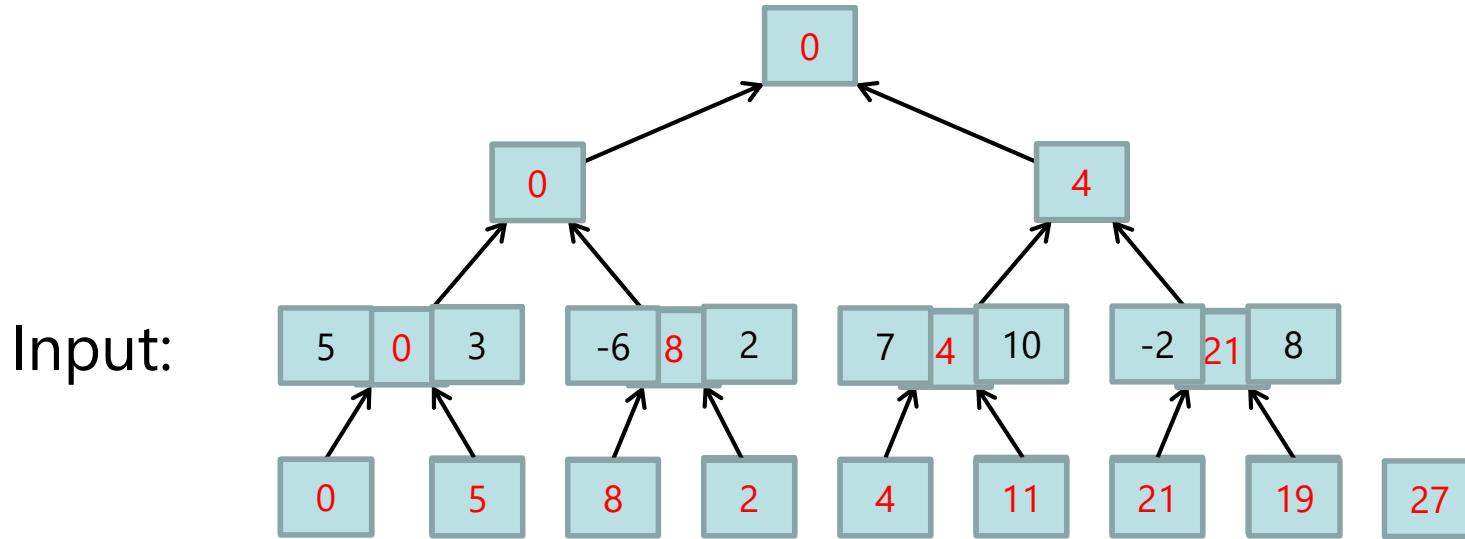
---

- Zweiter Durchgang
- Idee:
  - *Top-down-Berechnung zur Erzeugung aller Präfixsummen*
- Notation:
  - $\text{pre}[v]$  steht für die Summe in jedem Knoten  $v$
- $\text{pre}[\text{wurzel}] = 0$ 
  - $0$  ist das *neutrale Element* von  $+$ , d.h.  $x + 0 = x$  für alle  $x$
  - Bei einem anderen Operator muss das entsprechende neutrale Element gewählt werden (z.B.  $-\infty$  für MAX oder  $\infty$  für MIN)

# Parallele Präfixsumme

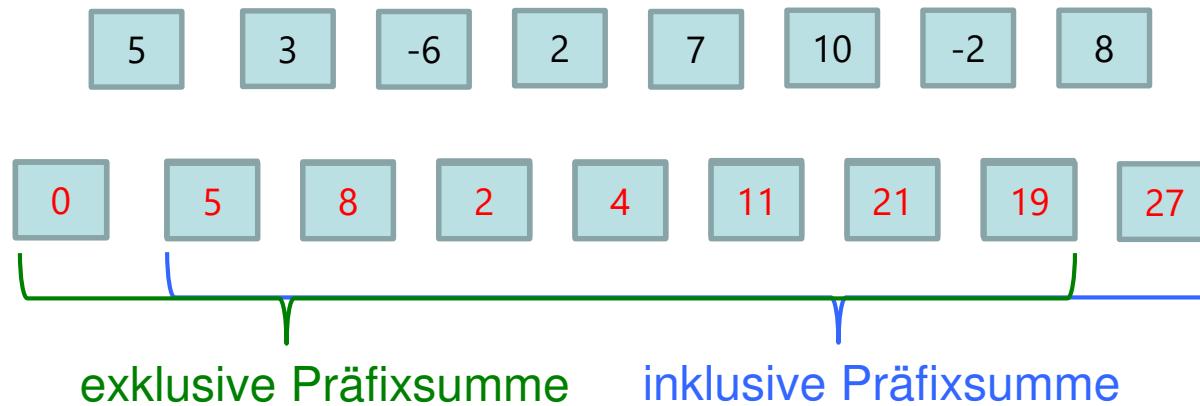
- Zweiter Durchgang: **top-down**

- $pre[v.right] = sum[v.left] + pre[v]$
- $pre[v.left] = pre [v]$



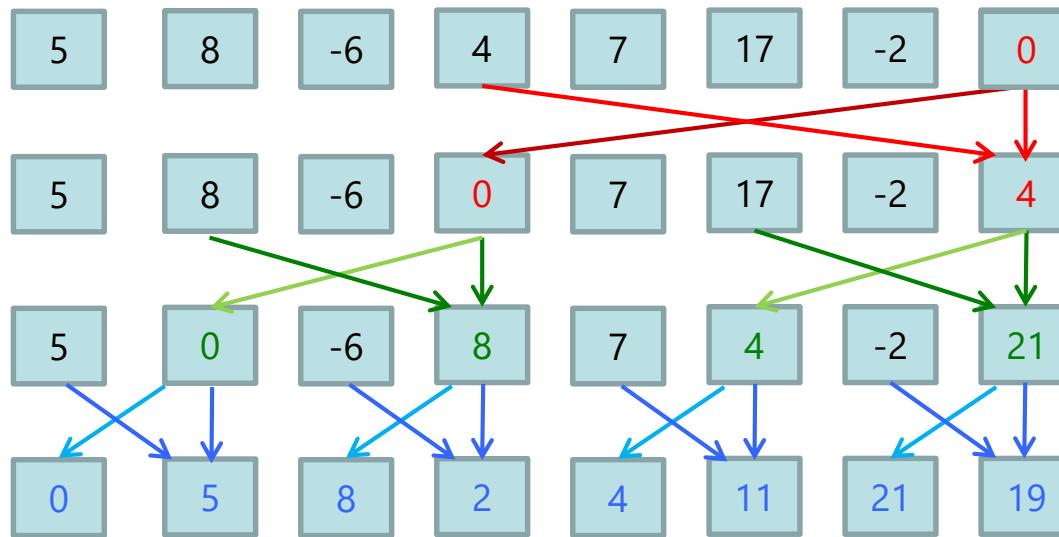
- Dies ist schon „fast“ die gewünschte Lösung!

# Inklusive vs. exklusive Präfixsumme



# Pseudocode

```
a[n-1] = 0
for d=log(n)-1; d>=0; d-- do
    for i=0; i<=n-1; i+=2d+1 in parallel
        temp = a[i + 2d - 1]
        a[i + 2d - 1] = a[i + 2d+1 - 1]
        a[i + 2d+1 - 1] = temp + a[i + 2d+1 - 1]
```



# Komplexität

---

- Der Algorithmus benötigt
  - Zeit:  $O(\log n)$
  - Prozessoren:  $n/2 = O(n)$ 
    - Dies lässt sich reduzieren auf  $O(n/\log n)$  **Übungsaufgabe!**

# Ist der Algorithmus korrekt?

---

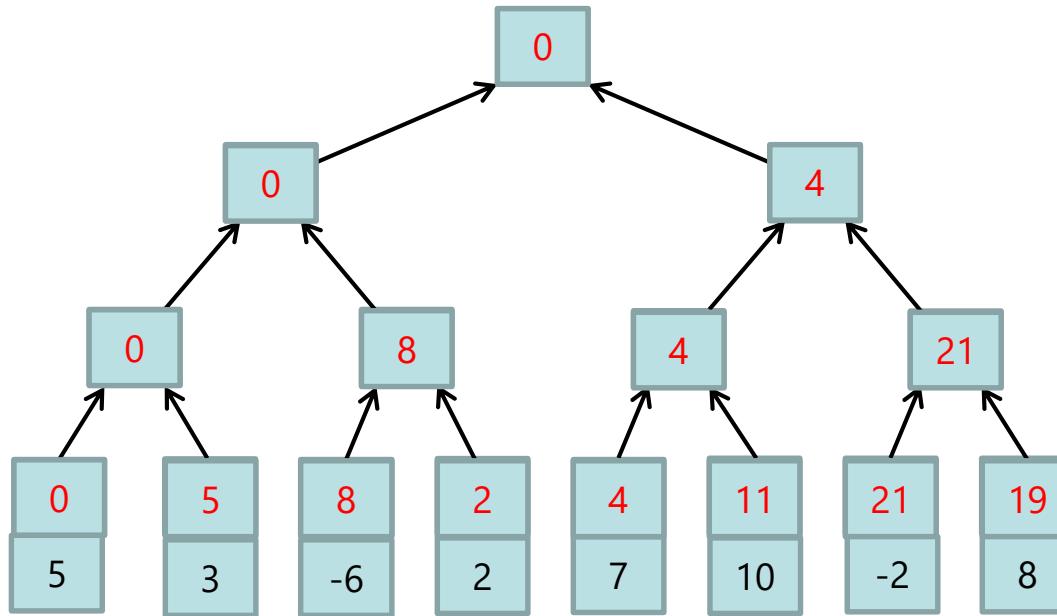
Definition:

Knoten  $x$  heißt **Vorgänger** von Knoten  $y$  wenn  $x$  bei einer Preorder-Traversierung (=Tiefensuche) im Baum **vor**  $y$  erreicht wird.

# Korrektheit

Lemma:

Nach dem zweiten Durchgang (top-down) enthält jeder Knoten die **Summe** der Werte **aller Blattknoten**, die im Ursprungsbaum seinen **Vorgängern** entsprechen.



# Korrektheit

---

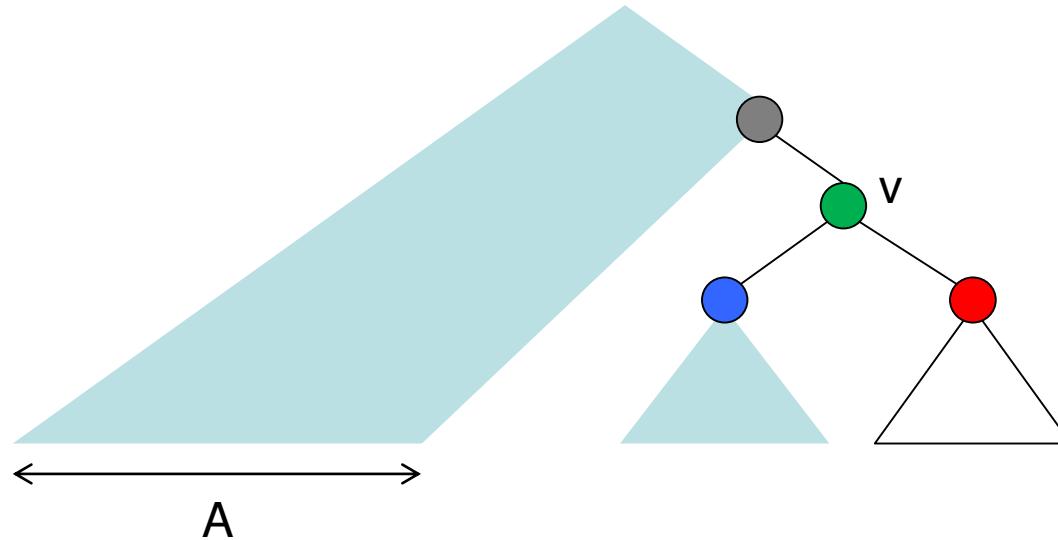
Lemma:

Nach dem zweiten Durchgang (top-down) enthält jeder Knoten die **Summe** der Werte **aller Blattknoten**, die im Ursprungsbaum seine **Vorgänger** sind.

Beweis (durch *strukturelle Induktion*):

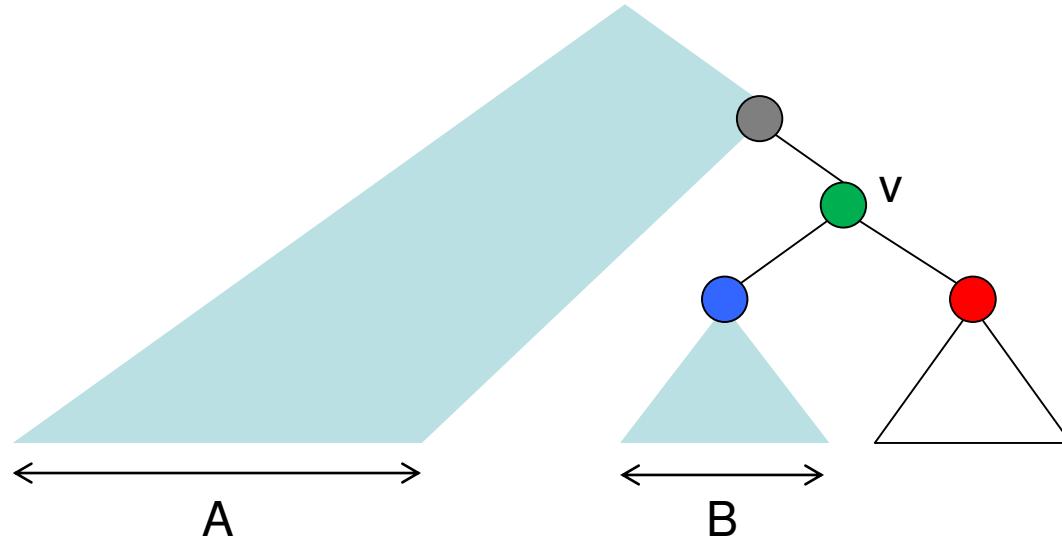
1. Für die **Wurzel** (mit Wert 0) gilt die Aussage, weil sie als erster besuchter Knoten keine Vorgänger hat.
2. Zu zeigen: Wenn die Aussage für **irgendeinen Knoten  $v$**  gilt, gilt er auch für seine **beiden Kinder  $v.left$  und  $v.right$** .

# Strukturelle Induktion



- Der linke Kindknoten von  $v$  hat genau dieselben Vorgänger-Blätter wie  $v$  selbst, nämlich die in Region A.
- Und nach Definition ist auch  $pre[v.left] = pre[v]$

# Strukturelle Induktion



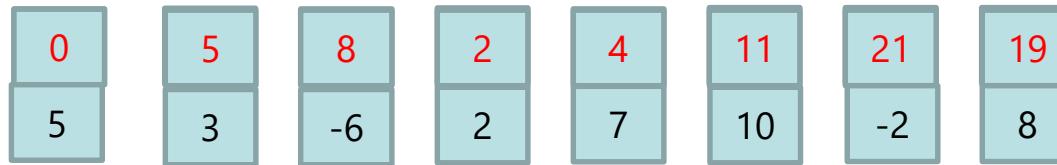
- Der **rechte Kindknoten** von  $v$  hat zwei Mengen von Vorgängern
  - Region A: entspricht  $pre[v]$
  - Region B:                entspricht  $sum[v.left]$
- Nach Definition ist  $pre[v.right] = pre[v] + sum[v.left]$

# Korrektheit

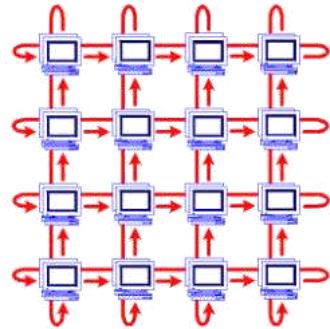
Lemma:

Nach dem zweiten Durchgang (top-down) enthält **jeder Knoten** die **Summe** der Werte **aller Blattknoten**, die im Ursprungsbaum seine **Vorgänger** sind.

Wenn es für jeden Knoten gilt, gilt es insbesondere für die Blätter.



Das entspricht genau dem, was wir zeigen wollen.



---

# Parallel Computing

Tobias Lauer

Hochschule Offenburg

---

# Theorie vs. Praxis

---

- „There's nothing more practical than a good theory“  
(Kurt Lewin)
- Unsere bisherigen Annahmen:
  - Zeitkomplexität wird in Abhängigkeit von der Prozessorkomplexität angegeben
  - Wir haben so viele Prozessoren „wie wir brauchen“
    - z.B.  $O(n / \log n)$ ,  $O(n)$ , ...
- Aber:
  - In der Praxis hat man i.d.R. immer nur eine konstante Zahl  $p$  von Prozessoren zur Verfügung

# Parallelisierung mit $p$ Prozessoren

---

- Es sei ...
  - $T_1$  die Zeit für die *sequenzielle* Ausführung des Algorithmus
  - $T_p$  die Zeit für die *parallele* Ausführung mit  $p$  Prozessoren
  - $T_\infty$  die Zeit für die *maximal parallele* Ausführung
- Dann gilt:
  - $T_p \geq T_1 / p$  (*Begrenzung durch linearen Speedup*)
    - » Der lineare Speedup wäre  $\Theta(p)$
  - $T_p \geq T_\infty$  (*Begrenzung durch maximalen Speedup*)
    - » Der max. mögliche Speedup ist  $T_1 / T_\infty$

# Brent's Theorem (auch: Brent's Law)

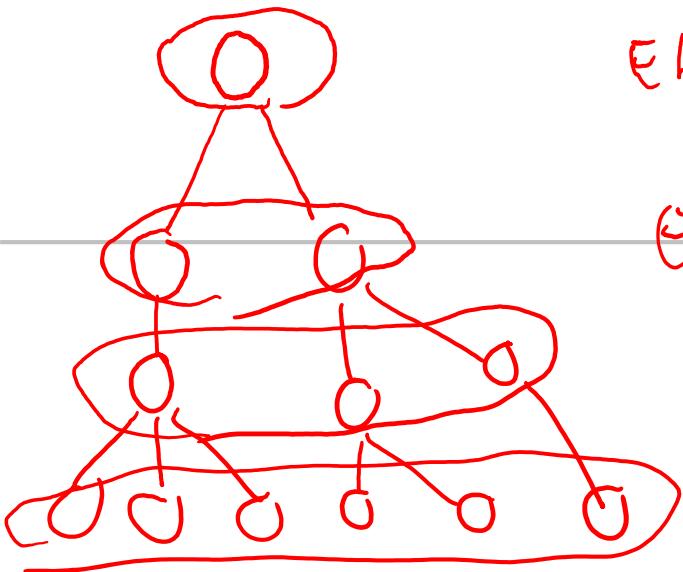
---

Ein Algorithmus  $A$  kann mit  $p$  Prozessoren parallel in Zeit

$$T_p \leq \frac{T_1}{p} + T_\infty$$

ausgeführt werden.

Beweis über Zuordnung von Jobs auf Prozessoren



Ebene 0  $\hat{=}$  Rechenschritt

$$m_1 = 1 \quad \text{Schritt}$$

Ebene 1

$$m_2 = 2 \quad \text{Schritt}$$

Ebene 3

$$m_3 = 3 \text{ Schritte}$$

Ebene 4  $m_4 = 6 \text{ Schritte}$

$$\overline{T}_1 = \sum_{i=1}^k m_i \quad , \text{ wobei } \underline{k} = \overline{T}_\infty$$

Sei  $T_p^i$  die Zeit zur Berechnung von Ebene  $i$  mit  $p$  Proz.

$$\overline{T}_p^i = \lceil \frac{m_i}{p} \rceil \leq \frac{m_i}{p} + 1$$

$$\overline{T}_p \leq \sum_{i=1}^k \frac{m_i}{p} + 1 = \underbrace{\frac{1}{p} \sum_{i=1}^k m_i}_{\overline{T}_1} + \sum_{i=1}^k 1$$

$$\overline{T}_p = \frac{\overline{T}_1}{p} + \overline{T}_\infty$$

□

# Beispiel

---

- Parallel Summierung (allg. *Reduktion*)
  - $T_1 = O(n)$
  - $T_\infty = O(\log n)$
  - $T_p \leq O(\log n) + n/p$

# Folgerung 1

---

- Aus den unteren Schranken  $T_p \geq T_1 / p$  und  $T_p \geq T_\infty$  und Brent's Theorem folgt:

$$\max\left(\frac{T_1}{p}, T_\infty\right) \leq T_p \leq 2 \cdot \max\left(\frac{T_1}{p}, T_\infty\right)$$

$$c \leq T_p \leq 2 \cdot c$$

# Folgerung 2

---

- Mit der Anzahl  $p = O(T_1 / T_\infty)$  an Prozessoren ist die (asymptotisch) optimale Zeit, d.h. ein maximaler Speedup erzielbar
- Beweis:
$$\begin{aligned}T_p &= \frac{T_1}{p} + T_\infty = \frac{o(T_1)}{o\left(\frac{T_1}{T_\infty}\right)} + T_\infty \\&= O\left(\frac{T_1}{\frac{T_1}{T_\infty}}\right) + O(T_\infty) = O(T_\infty)\end{aligned}$$

# Parallele Präfixsumme

---

- Problem:
  - *Gegeben:*
    - Array A von n Zahlen  
 $a_0, a_1, a_2, \dots, a_{n-1}$
    - Binärer assoziativer Operator  $\oplus$  (z.B. +, MAX, MIN)
  - *Gesucht:*
    - Array  
 $a_0, (a_0 \oplus a_1), (a_0 \oplus a_1 \oplus a_2), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-1})$

# Beispiel

---

- $\oplus$  sei die Addition und der Input sei

5, 3, -6, 2, 7, 10, -2, 8

- Dann ist der Output

5, 8, 2, 4, 11, 21, 19, 27

- Rechenzeit sequenziell:

$O(n)$

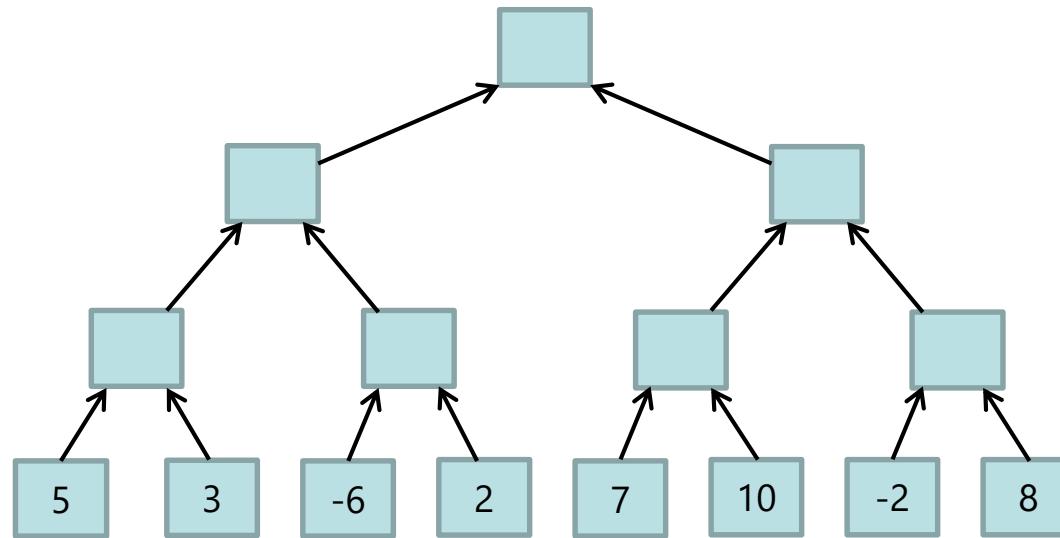
# Wie berechnet man das parallel?

---

- Zunächst: Warum überhaupt?
  - *oft benötigter Baustein für andere parallele Algorithmen*
  - „parallel primitive“

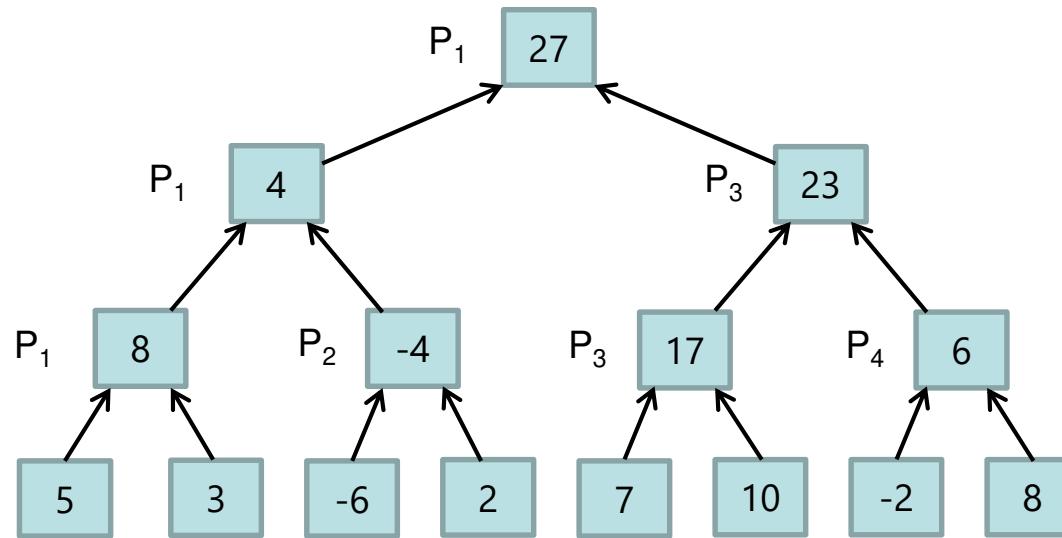
# Parallele Präfixsumme

- Zwei Durchgänge durch Array
  - *Bottom-up: Gesamtsumme berechnen*
  - *Top-down: Teilsummen verteilen*
- Repräsentation als Baum:



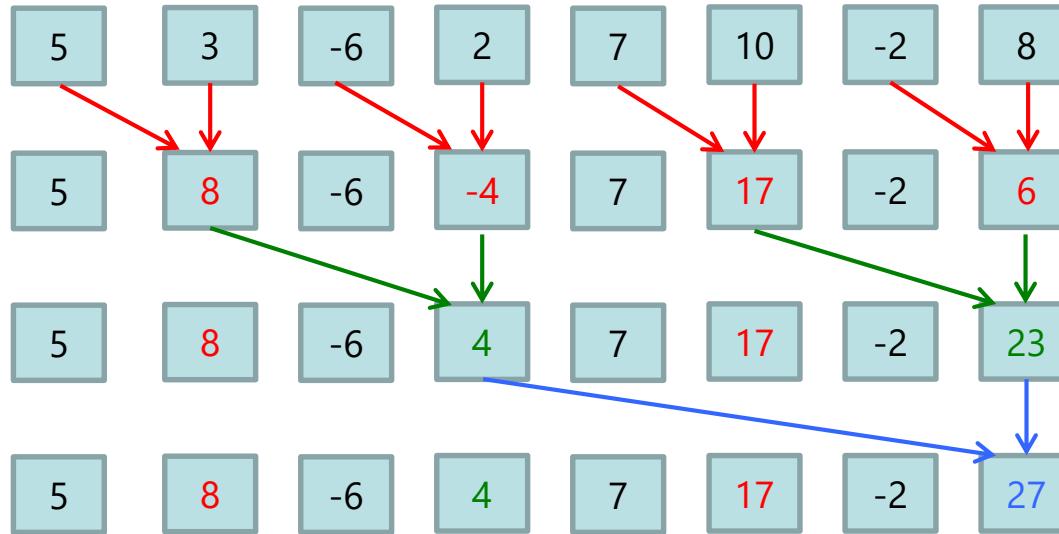
# Parallele Präfixsumme

- Erster Durchgang: **bottom-up**
  - $sum[v] = sum[v.left] + sum[v.right]$
  - Geht auch „in-place“



# Pseudocode

```
for d=0; d<=log(n)-1; d++ do
    for i=0; i<=n-1; i+=2d+1 in parallel
        a[i + 2d+1 - 1] = a[i + 2d - 1] + a[i + 2d+1 - 1]
```



# Parallele Präfixsumme

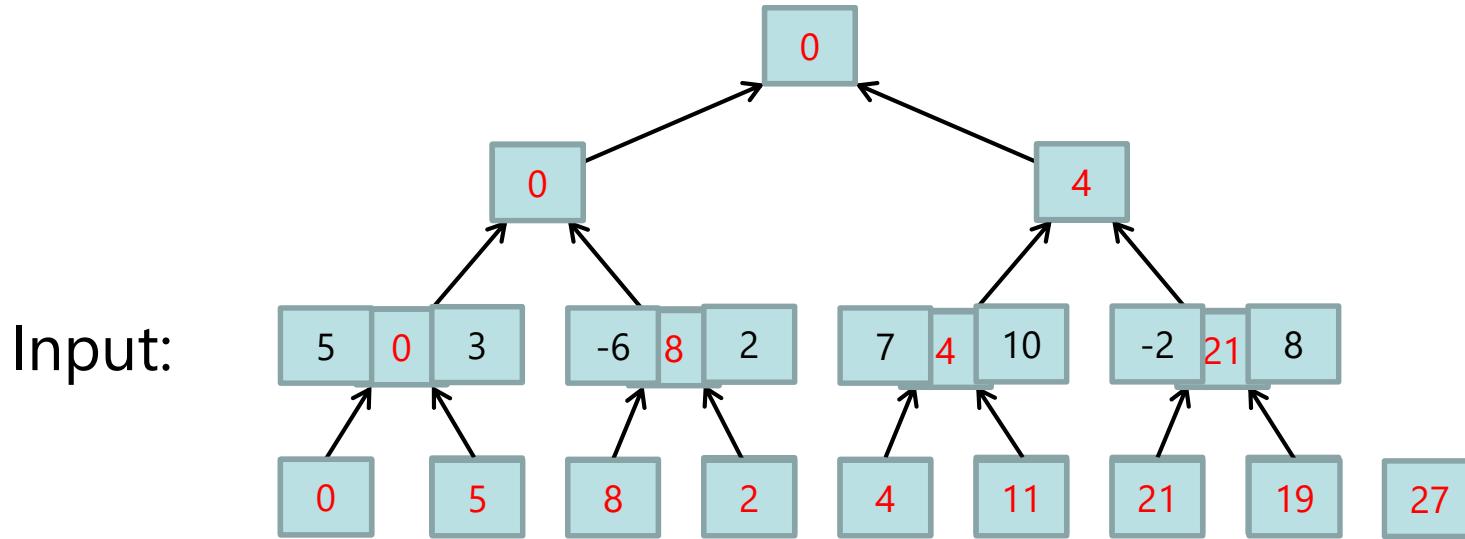
---

- Zweiter Durchgang
- Idee:
  - *Top-down-Berechnung zur Erzeugung aller Präfixsummen*
- Notation:
  - $\text{pre}[v]$  steht für die Summe in jedem Knoten  $v$
- $\text{pre}[\text{wurzel}] = 0$ 
  - $0$  ist das *neutrale Element* von  $+$ , d.h.  $x + 0 = x$  für alle  $x$
  - Bei einem anderen Operator muss das entsprechende neutrale Element gewählt werden (z.B.  $-\infty$  für MAX oder  $\infty$  für MIN)

# Parallele Präfixsumme

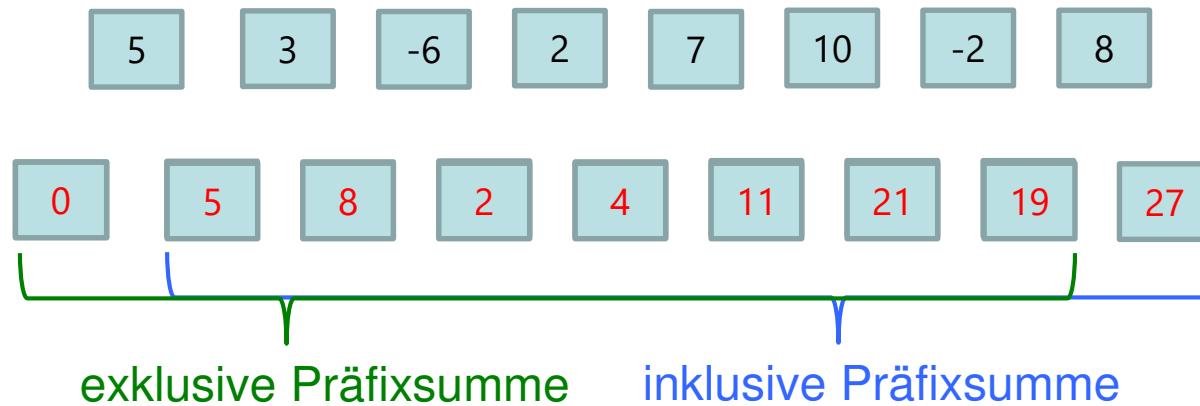
- Zweiter Durchgang: **top-down**

- $pre[v.right] = sum[v.left] + pre[v]$
- $pre[v.left] = pre [v]$



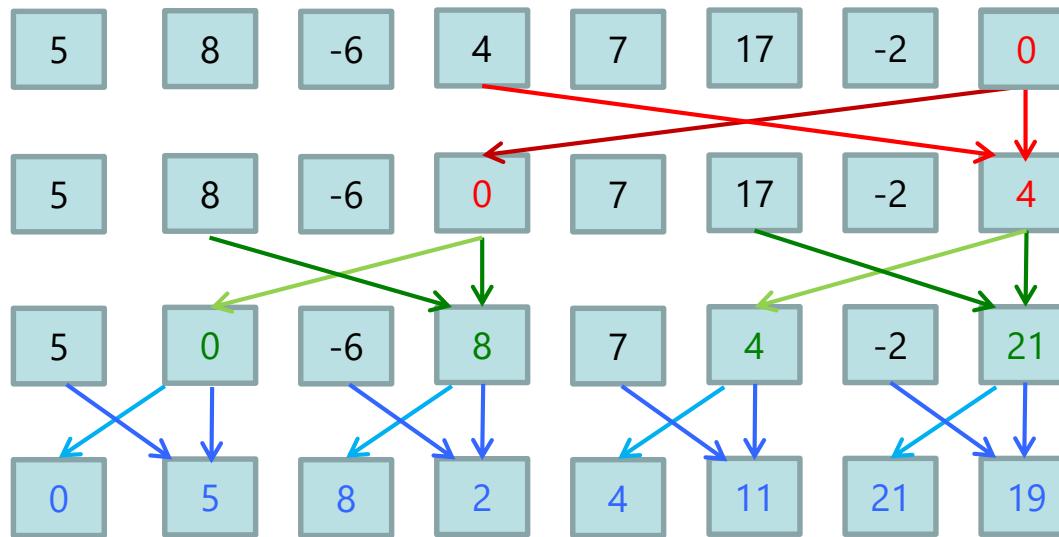
- Dies ist schon „fast“ die gewünschte Lösung!

# Inklusive vs. exklusive Präfixsumme



# Pseudocode

```
a[n-1] = 0
for d=log(n)-1; d>=0; d-- do
    for i=0; i<=n-1; i+=2d+1 in parallel
        temp = a[i + 2d - 1]
        a[i + 2d - 1] = a[i + 2d+1 - 1]
        a[i + 2d+1 - 1] = temp + a[i + 2d+1 - 1]
```



# Komplexität

---

- Der Algorithmus benötigt
  - Zeit:  $O(\log n)$
  - Prozessoren:  $n/2 = O(n)$ 
    - Dies lässt sich reduzieren auf  $O(n/\log n)$  **Übungsaufgabe!**

# Ist der Algorithmus korrekt?

---

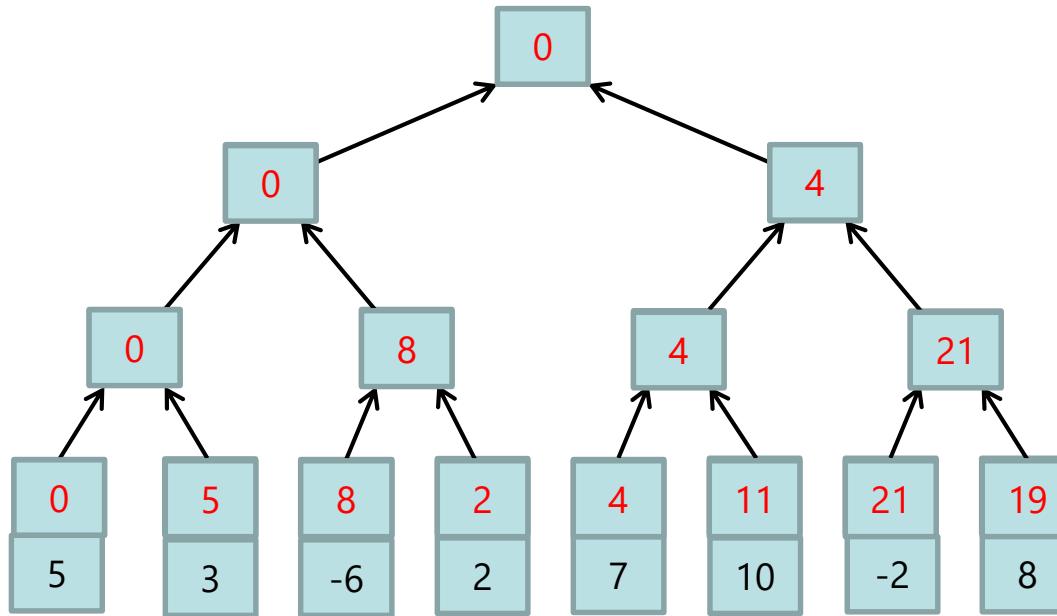
Definition:

Knoten  $x$  heißt **Vorgänger** von Knoten  $y$  wenn  $x$  bei einer Preorder-Traversierung (=Tiefensuche) im Baum **vor**  $y$  erreicht wird.

# Korrektheit

Lemma:

Nach dem zweiten Durchgang (top-down) enthält jeder Knoten die **Summe** der Werte **aller Blattknoten**, die im Ursprungsbaum seinen **Vorgängern** entsprechen.



# Korrektheit

---

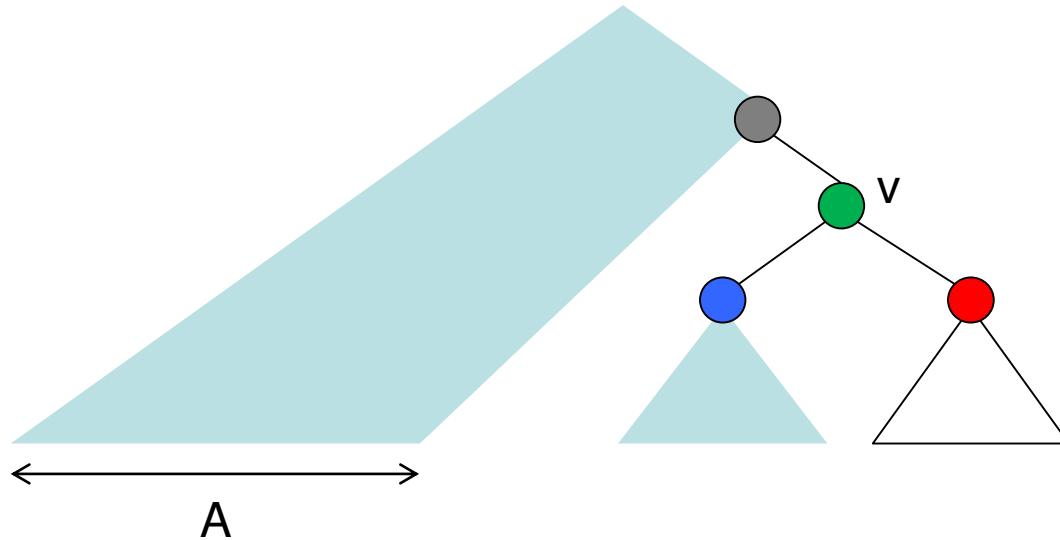
Lemma:

Nach dem zweiten Durchgang (top-down) enthält jeder Knoten die **Summe** der Werte **aller Blattknoten**, die im Ursprungsbaum seine **Vorgänger** sind.

Beweis (durch *strukturelle Induktion*):

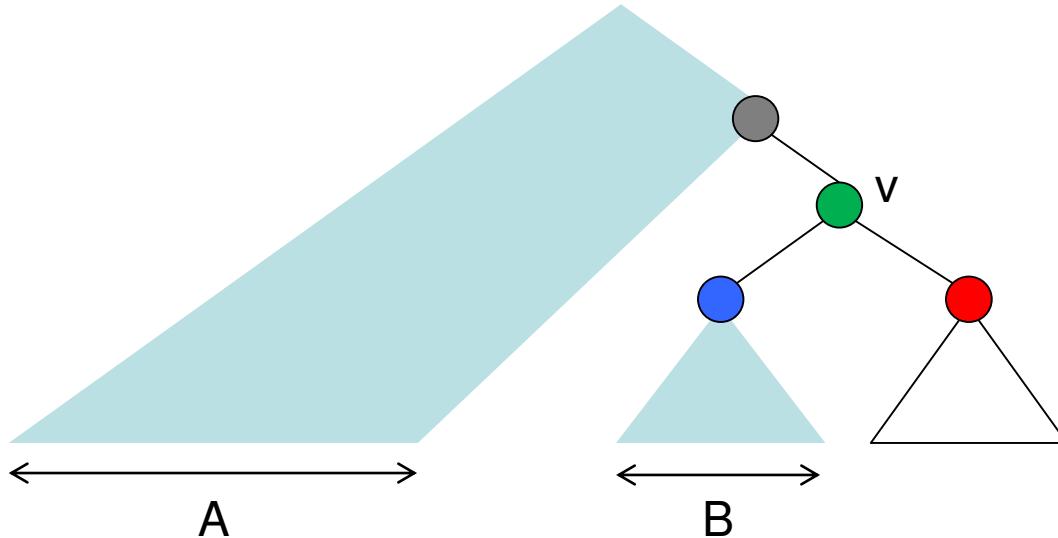
1. Für die **Wurzel** (mit Wert 0) gilt die Aussage, weil sie als erster besuchter Knoten keine Vorgänger hat.
2. Zu zeigen: Wenn die Aussage für **irgendeinen Knoten v** gilt, gilt er auch für seine **beiden Kinder v.left und v.right**.

# Strukturelle Induktion



- Der linke Kindknoten von  $v$  hat genau dieselben Vorgänger-Blätter wie  $v$  selbst, nämlich die in Region A.
- Und nach Definition ist auch  $pre[v.left] = pre[v]$

# Strukturelle Induktion



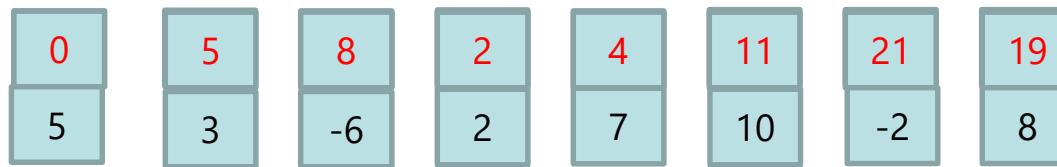
- Der **rechte Kindknoten** von  $v$  hat zwei Mengen von Vorgängern
  - Region A: entspricht  $pre[v]$
  - Region B:                entspricht  $sum[v.left]$
- Nach Definition ist  $pre[v.right] = pre[v] + sum[v.left]$

# Korrektheit

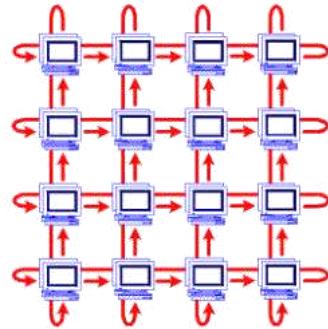
Lemma:

Nach dem zweiten Durchgang (top-down) enthält **jeder Knoten** die **Summe** der Werte **aller Blattknoten**, die im Ursprungsbaum seine **Vorgänger** sind.

Wenn es für jeden Knoten gilt, gilt es insbesondere für die Blätter.



Das entspricht genau dem, was wir zeigen wollen.



---

# Parallel Computing

Tobias Lauer

Hochschule Offenburg

---

# Themen

---

- Thread-Pools

# Parallelisierung: Tasks und Threads

---

- Aufgabe, abgeschlossene Arbeitseinheit
- (weitgehend) unabhängig von anderen Aufgaben zu erledigen
- Beispiele
  - *Bearbeitung einer Anfrage z.B. in einem Web-Server*
  - *Berechnung z.B. Bildbearbeitung*
- Task-Struktur einer Anwendung
  - *Tasks identifizieren*
  - *Tasks zerlegen:*
    - sequentiell abzuarbeiten
    - parallel abzuarbeiten

# Tasks und Threads

---

## Task

- Aufgabe

## Thread

- „Erlediger“ von Aufgaben

## Zuordnung kann auf unterschiedliche Weise erfolgen:

- *Sequenziell: ein Thread für alle Aufgaben*
- *Thread pro Task: für jede Aufgabe ein eigener Thread*
- *Thread-Pool: eine Menge von Threads erledigt eine Menge von Aufgaben*
- **Beachte:** Anwendung ist höchstens so schnell wie ihr am längsten dauernder Thread (vgl. Span = „critical path length“)

# Sequenzielle Zuordnung: 1 Thread

---

- Vorteile
  - *Einfach zu implementieren*
  - *Keine Synchronisierung im Code erforderlich*
  - *Kein Overhead für Thread-Verwaltung*
- Nachteile
  - *Schlechter Durchsatz (z.B. bei Web-Server)*
  - *Tasks müssen ggfs. ohne Not lange Zeit warten*
  - *Teile der Hardware liegen ggfs. unnötig brach*

# Eigener Thread pro Task

---

- Vorteile:
  - *Verbesserter Durchsatz*
  - *Hardware-Ressourcen werden genutzt*
- Nachteile:
  - *Code muss threadsicher sein*
  - *Overhead, vor allem bei vielen Tasks:*
    - Permanente Erzeugung/Vernichtung von Threads
    - Ressourcenverbrauch (z.B. Stack-Bereich)
    - Siehe Bsp. Merge-Sort, Editierdistanz, Heapify
  - *Mögliche Stabilitätsprobleme*

# Kompromiss: Thread-Pool

---

- Idee: benutze eine begrenzte Zahl von Threads, die nicht überschritten wird
- Vorteile beider Ansätze nutzen, Nachteile minimieren
  - nutzt Ressourcen (z.B. Multi-Core CPUs)
  - schont Ressourcen (z.B. Speicher für virtuellen Adressraum)
- Nachteil
  - Arbeit zur Verwaltung des Pools
  - Code muss threadsicher sein
- Das Executor-Framework in `java.util.concurrent` bietet Interfaces und Klassen für Thread-Pools

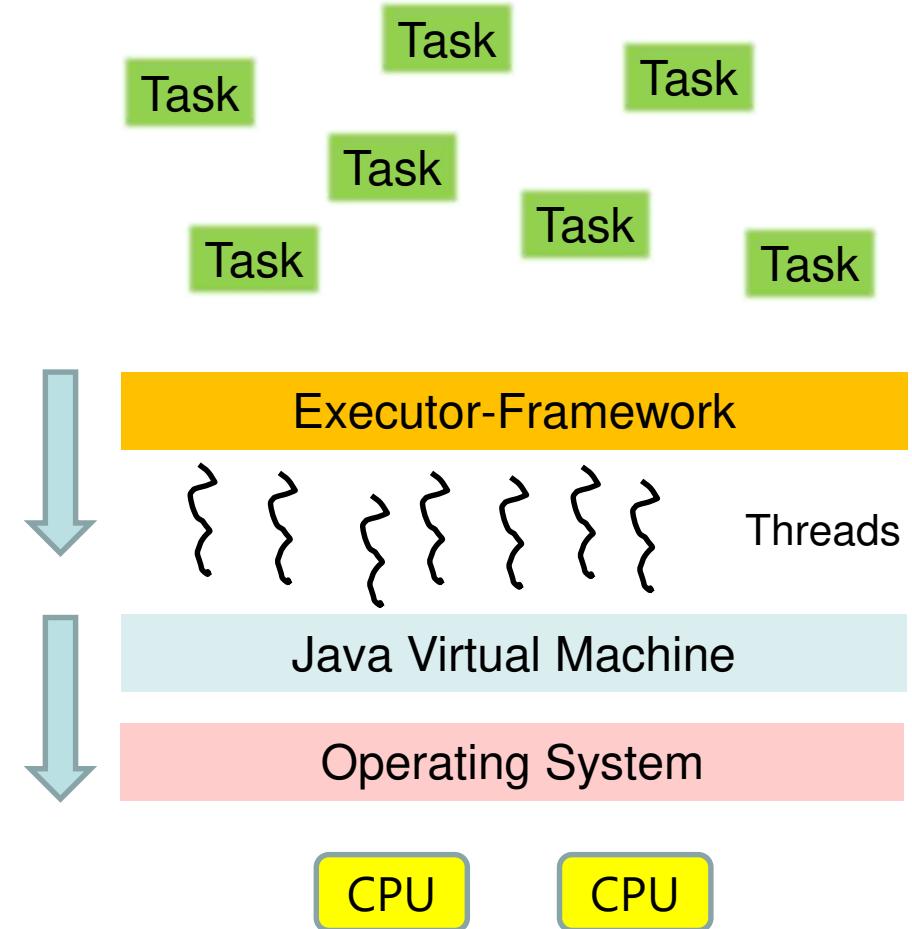
# Executor-Framework

---

- Interface, das die Ausführung von Tasks beschreibt
- Ein **Executor** (d.h. eine Klasse, die das Interface implementiert) führt Tasks mit Hilfe von Threads aus
  - *Kann Zuordnung in beliebiger Art vornehmen (Thread-Policy), z.B.*
    - Single-Threaded
    - Thread per Task
    - Thread-Pool
  - *Erzeugung und Wiederverwendung von Threads ist dem Executor überlassen*
- Vorteil: Entkopplung der Anwendung von der Thread-Policy
  - *Einfache Änderung der Thread-Policy möglich*
  - *Anwendung muss sich gar nicht um Thread-Policy kümmern*

# Executor-Framework als Zwischenschicht

- einfache standardisierte Mittel zur Erzeugung, Verwaltung und Einplanung von Threads
- unterstützt die Entkopplung
  - *der Aufgaben (Tasks) (anwendungsspezifisch)*
  - *der Konfiguration der Ausführung (abhängig von der Ablaufumgebung)*
- Für Softwareentwicklung der direkten Verwendung von Threads vorzuziehen



# Executor

---

- Executor:

- `public void execute(Runnable r)`

- Beispiel 1:

- *Ausführung jeder Task in einem eigenen Thread:*

```
class ThreadPerTaskExecutor implements Executor {  
    public void execute(Runnable r) {  
        new Thread(r).start();  
    }  
}
```

- Tasks müssen threadsicher programmiert sein!

# Executor

---

- Beispiel 2:

- Ausführung der Task im aufrufenden Thread:

```
class DirectExecutor implements Executor {  
    public void execute(Runnable r) {  
        r.run();  
    }  
}
```

- Wie ist es hier mit der Threadsicherheit von Tasks?

# Executor

---

- Beispiel 3:

- *Serielle Ausführung aller Tasks garantiert im selben Thread:*

```
public class OneThreadExecutor extends Thread implements Executor {  
    private final Queue<Runnable> tasks = new  
        LinkedBlockingQueue<Runnable>();  
  
    public void run() {  
        while (true)          // bzw. Bedingung für Beendigung  
            try {  
                tasks.take().run();  
            } catch(InterruptedException e) { ... }  
    }  
  
    public synchronized void execute(Runnable r) {  
        tasks.offer(r);  
    }  
}
```

# Executor Framework

---

- Comes with most important types of Executor
  - *ThreadPoolExecutor*
  - *ScheduledThreadPoolExecutor* für
    - **delayed** or
    - **periodic** execution of tasks
- Even better: **ExecutorService** **extends** **Executor**
  - *Additional methods for managing, e.g.*
    - **shutdown()**
    - **isTerminated()**
    - **submit()**
    - **...**

# ThreadPoolExecutor

---

- Components
  - Queue  $Q$  for incoming tasks
  - Threads: take tasks from  $Q$  and execute them
- Number  $t$  of threads in Pool
  - Core pool size: desired number of threads, only exceeded if necessary
  - Maximum pool size: maximum number of threads, will never be exceeded

# How it works

---

- New task comes in:
  - If  $t < \text{core pool size}$ :
    - Create a new thread for the task
  - If  $t \geq \text{core pool size}$  und  $t < \text{maximum pool size}$ 
    - Put task on Q
  - If  $t > \text{core pool size}$  und  $t < \text{maximum pool size}$  und Q voll
    - Create new thread
  - If  $t \geq \text{maximum pool size}$  und Q voll
    - ? (find out!)
- If  $t > \text{core pool size}$  and Q is empty
  - Terminate next idle thread

# Tasks with result

---

- So far: tasks for Executor must be **Runnables**
  - No (*explicit*) result – `public void run()`
  - Any *result* must be fetched manually (e.g. with *get* method)
- For tasks with result: **Callable<V>**
  - *Interface:* `public V call()`
  - Like *Runnable*, but returns *result* of type **V**
  - Can be executed by an Executor
  - Call with `executor.submit(task)`
- How to retrieve the result?

# Results

---

- Problem:
  - *Result has not been calculated yet at the time of task submission*
  - *Return value of `executor.submit(task)` must represent a future object of Type `V`*
- Solution: **Future<V>**
  - *Interface: `public V get()`*
  - *Represents the result (of Type `V`) of the execution
    - of a `Callable<V>`
    - by an Executor*
  - *Call: `future = executor.submit(task)`*

# Callable and Future

---

- Application
  - Submits a **Callable<V>** o an **ExecutorService**
  - Receives a **Future<V>** immediately
  - Result can be retrieved from Future
- Executor
  - Receives a **Callable<V>**
  - Returns a **Future<V>**
  - Computes the task with the help of threads
  - Hands the result to Future
- How to know when the result is there?

# Methods of Future

---

- `get()`
  - Blocks the calling thread until result is there
- `get(long timeout, TimeUnit unit)`
  - Waits for result or until time is up (whatever comes first)
- `isDone()`
  - `true` if and only if task has ended (no matter how!)
- `isCancelled()`
  - `true` if and only if task has been cancelled
- `boolean cancel(boolean b)`
  - Tries to cancel the corresponding task (with/without interruption)
  - `true` if and only if successful

# java.util.concurrent.atomic

---

- Atomic data types
  - `AtomicBoolean`
  - `AtomicInteger und AtomicIntegerArray`
  - `AtomicLong und AtomicLongArray`
  - `AtomicReference<V> und AtomicReferenceArray<E>`
  - ...
- Atomic methods for above types
  - `compareAndSet()`
  - `addAndGet()`
  - `addAndGet() und getAndAdd()`
  - `incrementAndGet() und getAndIncrement()`
  - ...

# java.util.concurrent.locks

---

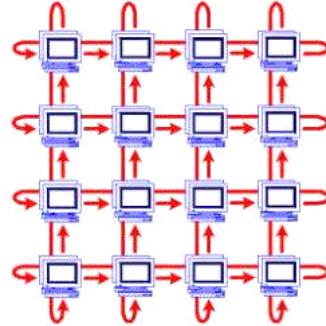
- Interfaces:
  - **Lock**
    - `tryLock(...)`
    - `lock()`
    - `unlock()`
  - **ReadWriteLock**
  - **Condition**
- Classes:
  - **ReentrantLock**
  - **ReentrantReadWriteLock**

# Synchronization classes

---

- **CountDownLatch**
  - *Barrier with countdown which must be counted down explicitly (using `countDown()`) Not reusable.*
- **CyclicBarrier**
  - *Barrier for a number of threads, reusable.*
- **Exchanger**
  - *Barriere for two threads to exchange data*
- **Semaphore**
  - *Implementation of a typical counting semaphore with key methods `acquire(...)` and `release(...)`*

.



---

# Parallele Programmierung und Algorithmen

---

Tobias Lauer

Hochschule Offenburg

# Themen

---

- Thread-Pools
  - *Nachteile von Standard-Threadpools*
  - *Fork/Join Framework*
- Parallel Algorithmen
  - *Parallele Maximumsbestimmung*

# Thread-Pools

---

- Thread-Pools des Executor-Frameworks **nicht** ideal für
  - *rekursive Tasks*
    - Potenziell hohe Threadzahl
    - Gefahr von Deadlocks (s. Merge-Sort) bei Begrenzung
  - *stark abhängige Tasks*
    - Entweder alle oder keine Tasks
    - Gefahr von Deadlocks, wenn Pool zu klein
- Ein Grund: Queuing aller Tasks in **einer** Schlange
  - *Reihenfolge ist oft entscheidend!*

# Alternative: Fork/Join Framework

---

- Geeignet für Algorithmen, die
  - aus *sehr vielen* Teilaktionen bestehen
    - die (pro „Ebene“) unabhängig voneinander bearbeitet werden können
  - möglichst *keinen Synchronisationscode* bzw. *blockierende Aktionen enthalten (außer Task-Subtask-Abhängigkeiten)*
- Typische Anwendungen
  - *Divide-and-Conquer*-Algorithmen
  - *Parallele Algorithmen auf Arrays*
    - Lohnt sich auf sehr großen Arrays, wenn man viele Prozessoren/Cores zur Verfügung hat

# Fork/Join: Task-Konstrukte

---

- **RecursiveAction**

- Klasse für rekursive Aktion (*ohne Ergebnisrückgabe*)
- Methode: **void compute()**
- Ruft (voneinander unabhängige) Unteraktionen gleicher Art auf
  - Dazu werden Instanzen der gleichen Klasse verwendet
- Start der Unteraktionen (*Einplanen in den Pool*) mit  
**void invokeAll(...Unteraktionen...)**
  - Methode wartet auf Return
- Alternative: asynchroner Start einer einzelnen Unteraktion  
**unteraktion.fork()**

# Beispiel

- Array-Increment: Erhöhe jedes Element eines Arrays um 1

```
class IncrementAction extends RecursiveAction {  
    final long[] array;  
    final int lo, hi;  
  
    IncrementAction(long[] array, int lo, int hi) {  
        this.array = array;  
        this.lo = lo; this.hi = hi;  
    }  
  
    protected void compute() {  
        if (hi - lo < THRESHOLD) {          // eine sinnvoll gewählte Grenze  
            for (int i = lo; i < hi; ++i) array[i]++;  
        } else {  
            int mid = (lo + hi) >>> 1;  
            invokeAll(new IncrementAction(array, lo, mid),  
                      new IncrementAction(array, mid, hi));  
        }  
    }  
}
```

# Fork/Join: Task-Konstrukte

---

- **RecursiveTask<V>**

- Klasse für rekursive Aktion *mit Ergebnisrückgabe* vom Typ **V**
- Methode: **V compute()**
- Ruft (voneinander unabhängige) Sub-Tasks gleicher Art auf
  - Dazu werden Instanzen der gleichen Klasse verwendet
- Start der Sub-Tasks (*Einplanen in den Pool*) mit  
**✓ ~~void invokeAll(... Sub-Tasks...)~~**
  - Methode wartet auf Ergebnisrückgabe
- Alternative: asynchroner Start einer einzelnen Unteraktion  
**subtask.fork()**
- Abholen der Ergebnisse  
**V join()**

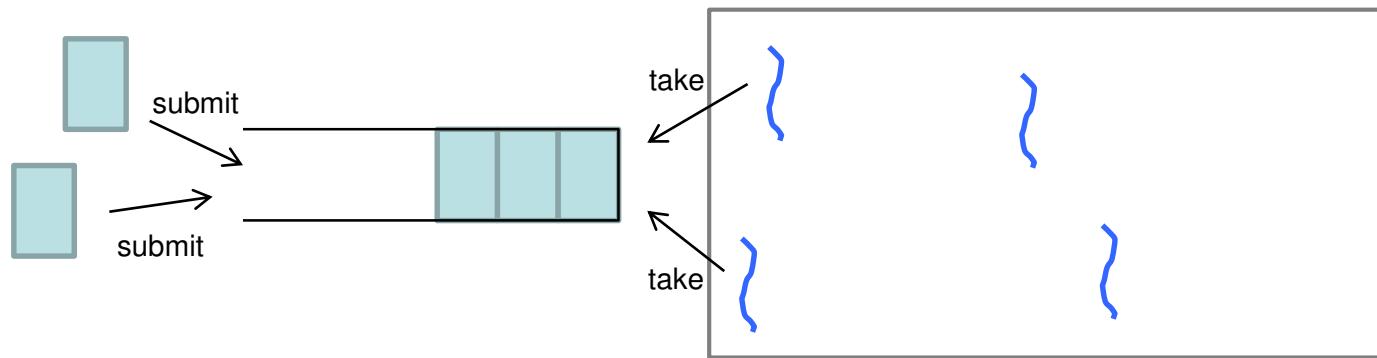
# Beispiel

---

```
class Fibonacci extends RecursiveTask<Integer> {  
    final int n;  
  
    Fibonacci(int n) {  
        this.n = n;  
    }  
  
    Integer compute() {  
        if (n <= 1) return n;  
        Fibonacci f1 = new Fibonacci(n-1);  
        f1.fork();  
        Fibonacci f2 = new Fibonacci(n-2);  
        return f2.compute() + f1.join();  
    }  
}
```

# Wie funktioniert der ForkJoinPool?

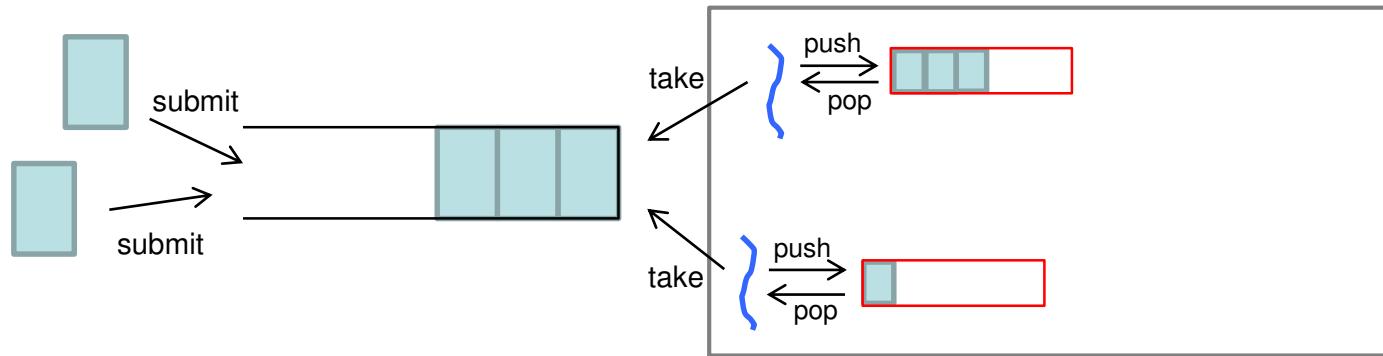
- Ein ForkJoinPool sieht „von außen“ aus wie jeder andere Thread-Pool:
  - *Es gibt eine Task-Queue, in der neue Tasks von Anwendungen abgeliefert werden...*



- *... und Threads, die sich diese Tasks abholen und ausführen*

# Thread-eigene Task-Queues

- Besonderheit:  
Jeder Thread hat zusätzlich eine **eigene Task-Queue**
  - Sub-Tasks der gerade ausgeführten Task werden hier eingefügt...



- ... und auch vorrangig vom Thread ausgeführt!

# Thread-eigene Queues

---

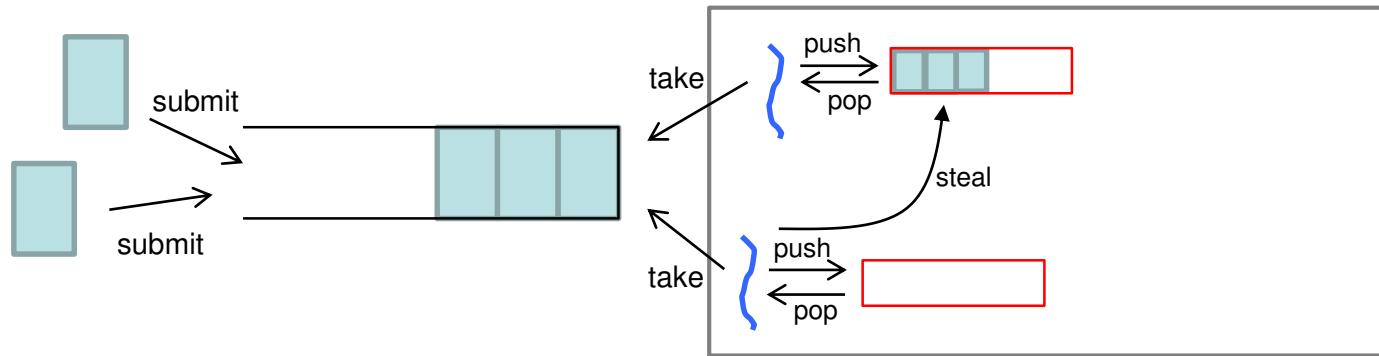
- Vorteile
  - Eine angefangene (Gesamt-)Aufgabe wird beendet, bevor der Thread eine komplett neue Aufgabe beginnt
  - Zentrale Queue wird nicht mit vielen kleinen Sub-Tasks belastet, sondern enthält nur (relativ) unabhängige Tasks
  - Dadurch weniger Konflikte (blocking) der Threads beim Zugriff auf die Queues (sowohl zentrale als auch thread-eigene)
- Realisierung der Zugriffe wie bei einem Stack (LIFO)
  - Zuletzt eingefügte Task wird zuerst bearbeitet
  - Subtasks kommen vor den übergeordneten Tasks an die Reihe
  - Vermeidung von Deadlocks

# Thread-eigene Queues

- ABER:
  - Eine angefangene (Gesamt-)Aufgabe wird beendet, bevor der Thread eine komplett neue Aufgabe beginnt
    - ABER: Was machen Threads, die schon fertig sind, während andere noch mit Sub-Taks beschäftigt sind? ?
  - Zentrale Queue wird nicht mit vielen kleinen Sub-Tasks belastet, sondern enthält nur (relativ) unabhängige Tasks
    - ABER: Was ist, wenn es nur 1 große Task gibt, die zerteilt wird (Bsp. Merge-Sort)??? ?
  - Dadurch weniger Konflikte (blocking) der Threads beim Zugriff auf die Queues (sowohl zentrale als auch thread-eigene)
    - ABER: Gefahr arbeitsloser Threads, weil zentrale und eigene Queue leer sind! ?

# Arbeitslose Threads

- Wenn die eigene Queue eines Threads leer ist...
  - Schaue zuerst in den *Task-Queues anderer Threads* nach...



- ... und nimm diesen – wenn möglich – Arbeit ab (work stealing).

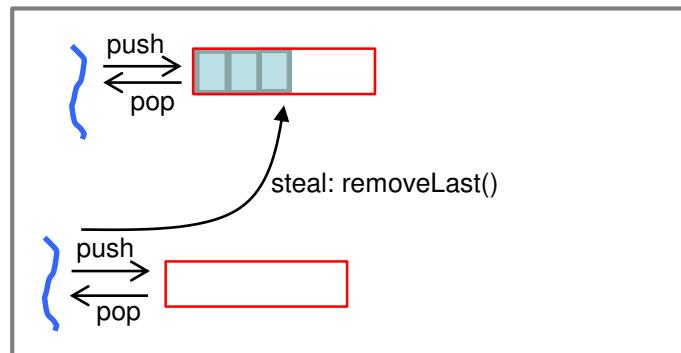
# Work stealing

---

- Vorteile:
  - Eine angefangene (Gesamt-)Aufgabe wird erst beendet, bevor – auch von anderen Threads – eine komplett neue Aufgabe begonnen wird
  - Threads erledigen auch feingranulare Tasks arbeitsteilig
- Möglicher Nachteil:
  - Zugriffskonflikte mit dem „Besitzer“-Thread der Queue
  - Feingranulare Tasks sind priorisiert (LIFO), daher viele Zugriffe wahrscheinlich
- Dies kann man verhindern:
  - durch Implementierung der **thread-eigenen Queue als Deque**.

# Deque (double ended queue)

- Erlaubt Operationen am Beginn (first) und am Ende (last)
  - Besitzer-Thread arbeitet nur am Beginn der Queue
    - push: addFirst(...)
    - pop: removeFirst()
  - Fremde Threads entfernen nur am Ende
    - steal: removeLast()



# Fazit: Fork/Join

---

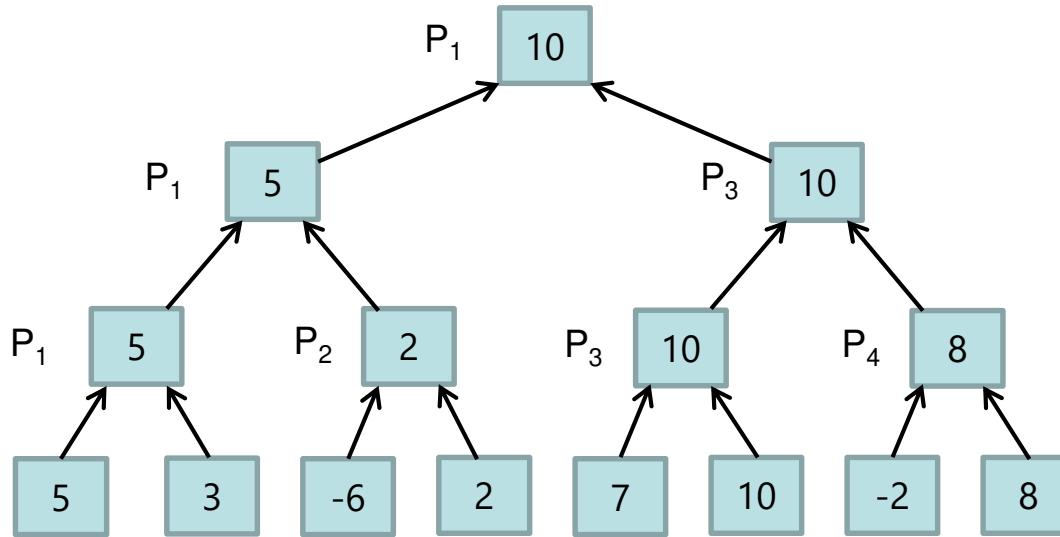
- Nicht unbedingt effizienter als ThreadPoolExecutor!
  - *Zusätzlicher Overhead* für Verwaltung von Queues und Work-Stealing
  - Wenn die Eingabe-Tasks sehr *ausbalanciert* sind, findet oft *kein Work-Stealing* statt.
  - Wenn das von Vornherein bekannt ist, kann man mit „normalem“ Thread-Pool oder manuellem Threading effizienter sein
- Work-Stealing sorgt für feiner granulare Lastverteilung
  - Gut geeignet, wenn eine große Task einen Thread auslasten würde, während andere Threads warten müssen
  - z.B. bei *ungleich verteilt*em Input, der *weiter zerlegt* werden kann
  - Fork/Join vorzuziehen bei *rekursiv* formulierten Tasks

# Parallel maximum

---

- Problem:  
Finde in einem Array  $A$  von  $n$  Zahlen die größte Zahl bzw. den Index (Position) der größten Zahl.
- Sequenzielle Lösung:  $O(n)$
- Behauptung: Wir wissen bereits, wie man das Maximum von  $n$  Zahlen parallel in Zeit  $O(\log n)$  bestimmen kann
  - $\text{MAX}$  ist wie  $+$  ein binärer assoziativer Operator
  - Daher kann derselbe Algorithmus wie für die einfache Summe verwendet werden, nur mit  $\text{MAX}$  statt  $+$

# Parallel maximum



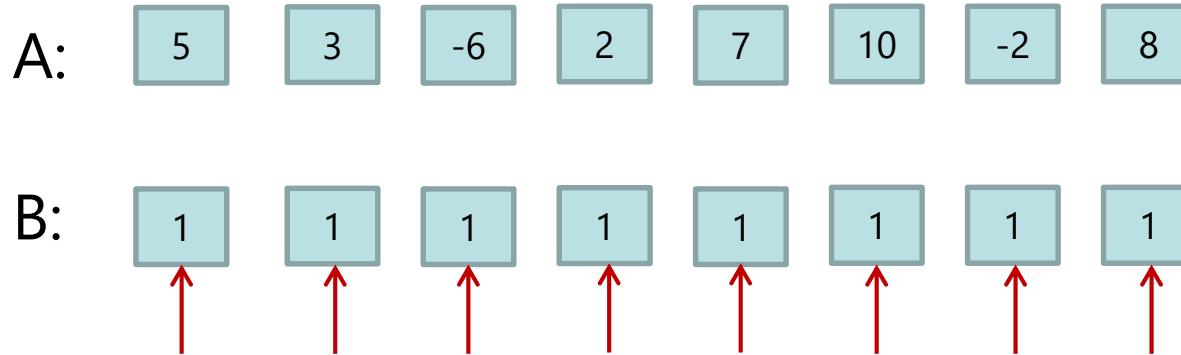
- Zeit  $T(n) = O(\log n)$
- Geht es noch schneller?

# Parallel maximum

---

- Behauptung:  
Man kann MAX sogar in Zeit  $O(1)$  bestimmen!
- Allerdings:  
Wir brauchen dafür viele Prozessoren (und extra Platz)

# Parallel maximum



- Start:
  - Erstelle ein Array **B** der Länge  $n$
  - Initialisiere es mit  $\mathbf{B[k]} = 1$  für alle  $k$
  - Das geht in  $O(1)$ , und wir brauchen „nur“  $n$  Prozessoren

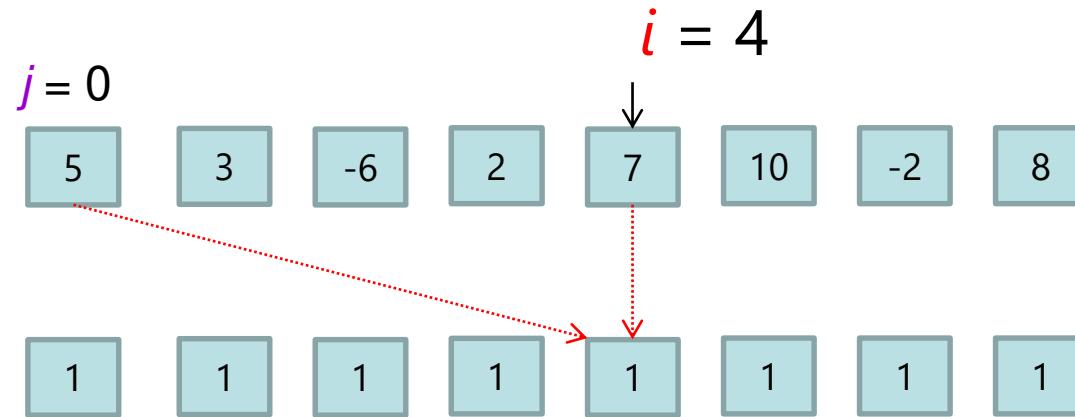
# Parallel maximum

---

- Schritt 1
  - Benutze  $n$  Prozessoren für jedes Element  $A[i]$
  - Wir betrachten ein Element  $A[i]$  und seine  $n$  Prozessoren  $P_{i,0}, P_{i,1}, \dots, P_{i,j}, \dots, P_{i,n-1}$
  - Jeder Prozessor  $P_{ij}$  vergleicht  $A[i]$  mit  $A[j]$ :  

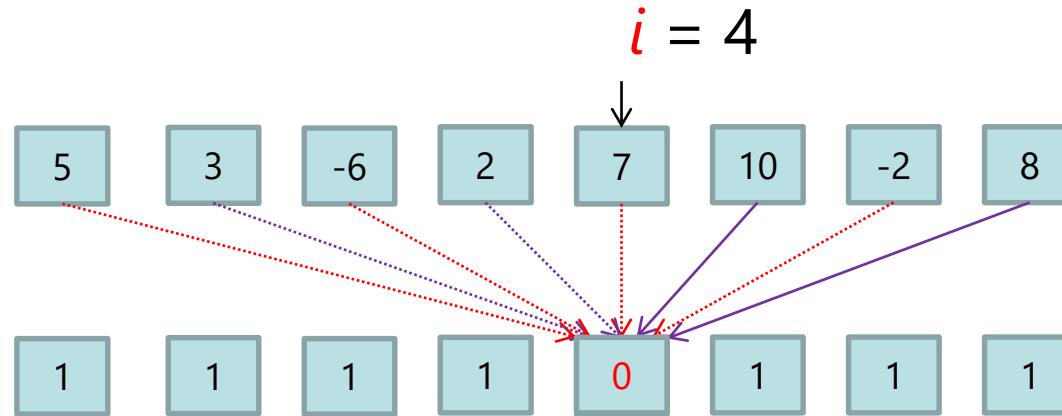
```
if (A[i] < A[j])
    B[i] = 0
else
    // do nothing
```

# Beispiel: $P_{4,0}$



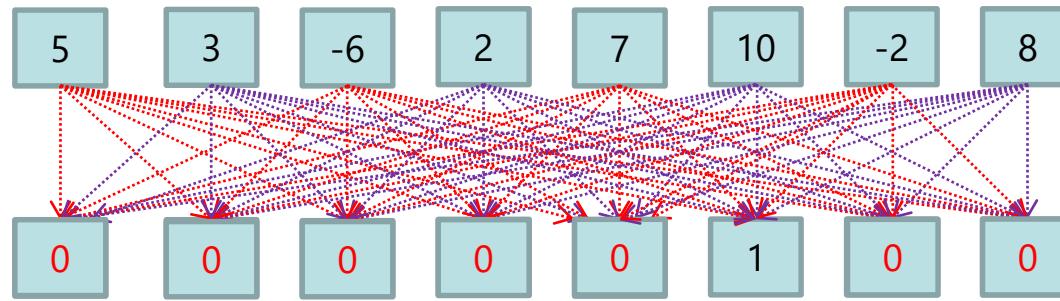
Führe dies parallel für alle  $j$  aus.

# Beispiel: $P_{4,j}$ für alle $j$



Führe dies auch parallel für alle  $i$  aus.

# Beispiel: alle $P_{i,j}$



$\Rightarrow n^2$  Prozessoren ...  
||||

# Parallel maximum

---

- Schritt 2:
  - Beobachtung:  
*Am Ende von Schritt 1 ist **B[k]** = 1 genau dann, wenn **A[k]** das maximale (bzw. ein maximales) Element ist*
  - Verwende **n** Prozessoren, einen für jeden Eintrag in **B**  

```
if B[i] == 1
    result_index = i      (oder result_max = A[i])
```

*(je nachdem, was als Antwort erwartet wird)*
- Beobachtung: In beiden Schritten ist ggfs. simultanes Schreiben (concurrent write) erforderlich

# Simultanes Schreiben

---

- Kann es Schreibkonflikte geben?
  - In Schritt 1:

Wenn **simultan** geschrieben wird, schreiben **alle** Prozessoren **dasselbe** („unechte“ race condition)  
→ möglich ab Common CRCW
  - In Schritt 2:

Da der **Index** (Position) des Maximums geschrieben wird, könnten verschiedene Threads unterschiedliche Dinge schreiben  
→ möglich nur ab Arbitrary CRCW (echte race condition)
  - Wenn jedoch direkt der **Wert** des Maximums geschrieben wird, geht auch dieser Schritt mit Common CRCW

# Komplexität

---

- Zeit:  $T(n) = O(1)$ 
  - *Besser geht's nicht!* ☺
- Prozessoren:  $p(n) = O(n^2)$ 
  - *Gar nicht gut!* ☹
- Kosten:  $c(n) = O(n^2)$ 
  - *Nicht cost-optimal* ☹

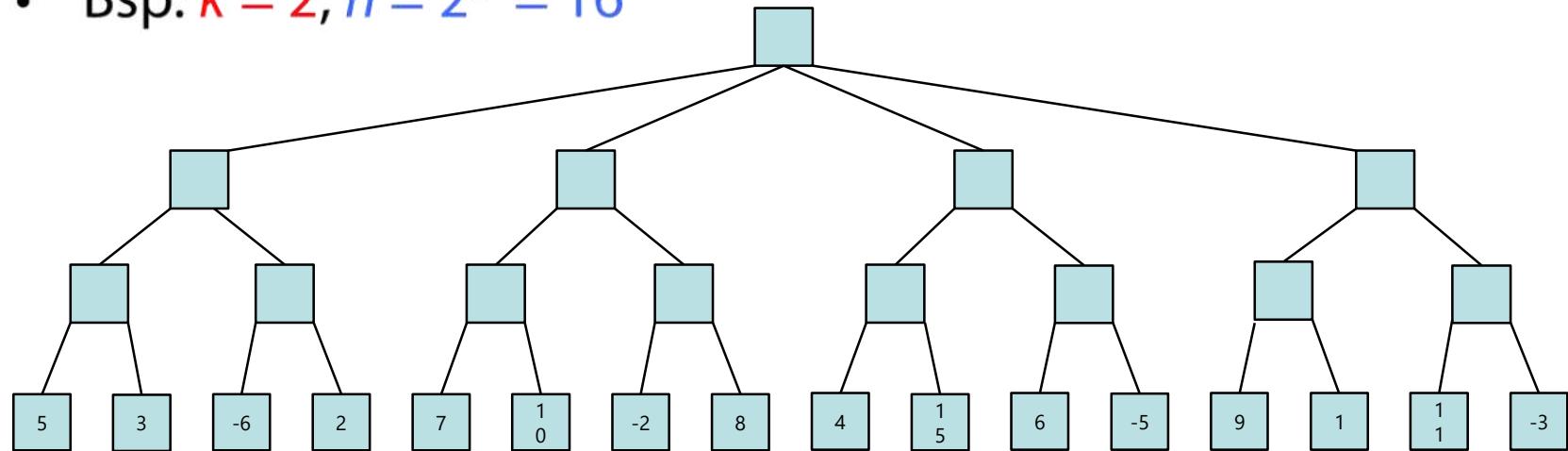
# Geht es besser?

---

- Ziele:
  - Realistischere Zahl von Prozessoren (weniger als  $n$ )
  - Trotzdem Laufzeit besser als  $O(\log n)$
  - Cost-Optimalität
- Ansatz:
  - Kombiniere die beiden bisherigen Algorithmen
    1.  $T(n) = O(\log n)$                        $c(n) = O(n)$
    2.  $T(n) = O(1)$                              $c(n) = O(n^2)$
  - Ziel
    - $T(n) = O(\log \log n)$                        $c(n) = O(n)$

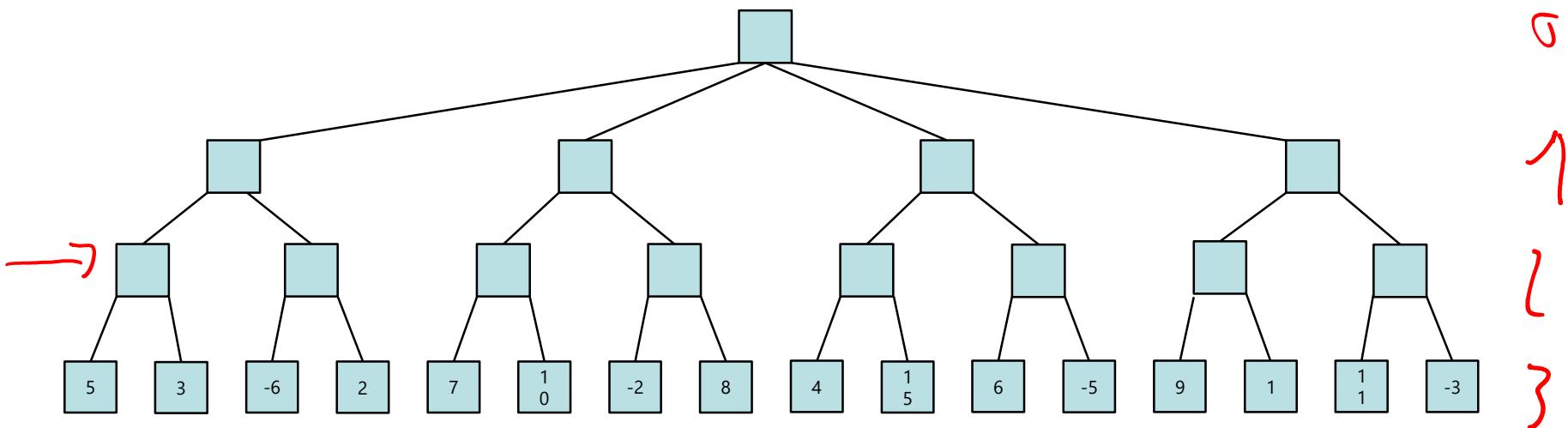
# Algorithmus mit Zeit $O(\log \log n)$

- Komplexerer Baum zur Repräsentation
- Wir nehmen an:  $n = 2^{2^k}$
- Jeder Knoten in Level  $i$  hat  $2^{2^{k-i-1}}$  Kinder für  $0 \leq i < k$ 
  - Die Wurzel des Baums (Level 0) hat also  $2^{2^{k-1}} = n^{1/2} = \sqrt{n}$  Kinder
- Jeder Knoten in Level  $k$  hat 2 Kinder (Blätter)
- Bsp:  $k = 2, n = 2^{2^2} = 16$



# Algorithmus mit Zeit $O(\log \log n)$

- Bsp:  $k = 2$ ,  $n = 2^{2^2} = 16$



- Tiefe des Baums:  $k + 1$
- Da  $n = 2^{2^k}$ , gilt:  $k = O(\log \log n)$
- Die Anzahl der Knoten auf Level  $i$  ist  $2^{2^k - 2^{k-i}}$  für  $0 \leq i \leq k$

# Algorithmus

---

- Gehe Level für Level vor
  - *Starte bei den Knoten direkt über den Blättern*
  - *In jedem Level:*
    - berechne für **jeden Knoten** das **Maximum aller Kinderknoten** in konstanter Zeit mit Hilfe des  **$O(1)$**  Algorithmus
  - *Berechne damit parallel den kompletten Level in konstanter Zeit*
    - Wichtig ist nur: Wieviele Prozessoren braucht man dafür?
- Dann ist die **Gesamtzeit** proportional zur Zahl der Level (Tiefe des Baums), also in  **$O(\log \log n)$**

# Kosten pro Ebene

---

- Jeder Knoten auf Level  $i$  hat  $c = 2^{2^{k-i-1}}$  Kinder
- Der Prozessoraufwand pro Knoten ist  $O(c^2) = O((2^{2^{k-i-1}})^2)$
- Auf Level  $i$  gibt es insgesamt  $2^{2^k - 2^{k-i}}$  Knoten, die wir parallel berechnen wollen ( $\rightarrow$  Zeit  $O(1)$  pro Level)
- Das heißt, die Gesamtkosten  $C_i(n)$  für Level  $i$  sind

$$O((2^{2^{k-i-1}})^2) \cdot 2^{2^k - 2^{k-i}} = O(2^{2^k}) = O(n)$$

- Beobachtung: Die Kosten  $C_i(n)$  sind von  $i$  unabhängig, also für jeden Level gleich!

# Nebenrechnung

---

$$\begin{aligned}& (2^{2k-i-1})^2 \cdot 2^{2k-2k-i} \\&= 2^{2 \cdot 2k-i-1} \cdot 2^{2k-2k-i} \\&= 2^{2k-i-1+1} \cdot 2^{2k-2k-i} \\&= 2^{2k-i} \cdot 2^{2k-2k-i} \\&= 2^{2k-i+2k-2k-i} \\&= 2^{2k} \\&= n\end{aligned}$$

# Anzahl Prozessoren

- Starte von oben
  - Ebene 0 (Wurzel)
    - 1 Knoten mir  $n^{1/2}$  Kindern  $\rightarrow 1 \cdot (n^{1/2})^2 = n$  Prozessoren
  - Ebene 1
    - $n^{1/2}$  Knoten mit je  $n^{1/4}$  Kindern  $\rightarrow n^{1/2} \cdot (n^{1/4})^2 = n$  Prozessoren
  - Ebene 2
    - $n^{3/4}$  Knoten mit je  $n^{1/8}$  Kindern  $\rightarrow n^{3/4} \cdot (n^{1/8})^2 = n$  Prozessoren
  - Ebene 3
    - $n^{7/8}$  Knoten mit je  $n^{1/16}$  Kindern  $\rightarrow n^{7/8} \cdot (n^{1/16})^2 = n$  Prozessoren
  - ...
  - Ebene über den Blättern:
    - $n/2$  Knoten mit 2 Kindern  $\rightarrow$  geht mit  $n/2$  Prozessoren

# Beispiel 1

---

- $n = 16$  ( $k = 2$ )
- Starte von unten
  - *Ebene über den Blättern:*
    - 8 Knoten/Prozessoren für Maximum von je 2 Blättern
  - *Ebene darüber:*
    - 4 Knoten mit je 2 Kindern → 16 Prozessoren  
(4 Prozessoren)
  - *Ebene darüber:*
    - 1 Knoten mit 4 Kindern → 16 Prozessoren

# Beispiel 2

---

- $n = 256$  ( $k = 3$ )
- Starte von unten
  - *Ebene über den Blättern:*
    - 128 Knoten à 2 Kinder (Blätter) → 128 Prozessoren
  - *Ebene darüber:*
    - 64 Knoten à 2 Kinder → 256 Prozessoren  
(oder: 64 Prozessoren)
  - *Ebene darüber:*
    - 16 Knoten à 4 Kinder → 256 Prozessoren
  - *Ebene darüber*
    - 1 Knoten à 16 Kinder → 256 Prozessoren

# Gesamtkosten

---

- Bei  $O(\log \log n)$  Ebenen mit je Cost  $O(n)$  ist der Gesamtaufwand (Kosten)

$$c(n) = O(n \log \log n) \quad \rightarrow \text{nicht cost-optimal}$$

- Verbesserung durch *Reduzierung des Inputs (algorithmic cascading)*
  - Kennen wir schon von Summe, Parallel Prefix etc.*
  - Unterteile den Input zunächst in geeignet große Teile*
  - Berechne für jedes Teilstück sequenziell das Maximum (aber alle Teilstücke parallel)*
  - Auf dem Ergebnis benutze den gerade vorgestellten Algorithmus*

# Phase 1: Input-Reduzierung

- Was ist eine geeignete Unterteilungsgröße?
  - Da wir in  $O(\log \log n)$  Zeit bleiben wollen, nehmen wir Stücke der Größe  $\log \log n$ .
  - Dafür brauchen wir  $\frac{n}{\log \log n}$  Prozessoren
  - Die Zahl der Kandidaten reduziert sich also auf  $\frac{n}{\log \log n}$
  - Die Zeit dafür ist in  $O(\log \log n)$
  - Die Arbeit ist in  $O(n)$ .

Erinnerung: Brent's Theorem:  $T_p \leq \frac{T_1}{p} + T_\infty$

Folgerung: Mit der Anzahl  $p = O(T_1 / T_\infty)$  an Prozessoren ist die (asymptotisch) optimale Zeit, d.h. ein maximaler Speedup erzielbar!

# Phase 2

---

- Vorheriger Algorithmus (komplexer Baum)
  - Benutze den Algorithmus für die verbleibenden  $n'$  Kandidaten, wobei  $n' = \frac{n}{\log \log n}$
  - Die *Gesamtarbeit* dafür ist
$$O(n' \log \log n') = O\left(\frac{n}{\log \log n} \log \log n'\right) = O(n)$$
  - Der *Zeitaufwand* ist
$$O(\log \log n') = O(\log \log n)$$

# Gesamtaufwand

---

- Phase 2:
  - Cost  $O(n)$
  - Time  $O(\log \log n)$
- Phase 1:
  - Cost  $O(n)$
  - Time  $O(\log \log n)$

Fazit (Theorem):

Das Maximum von  $n$  Elementen kann in **Zeit  $O(\log \log n)$**  und **Cost  $O(n)$**  auf einer **CRCW PRAM** berechnet werden.  
(Man kann zeigen, dass dies ist auch das Optimum ist.)

# Parallele Algorithmen ab Java 1.8

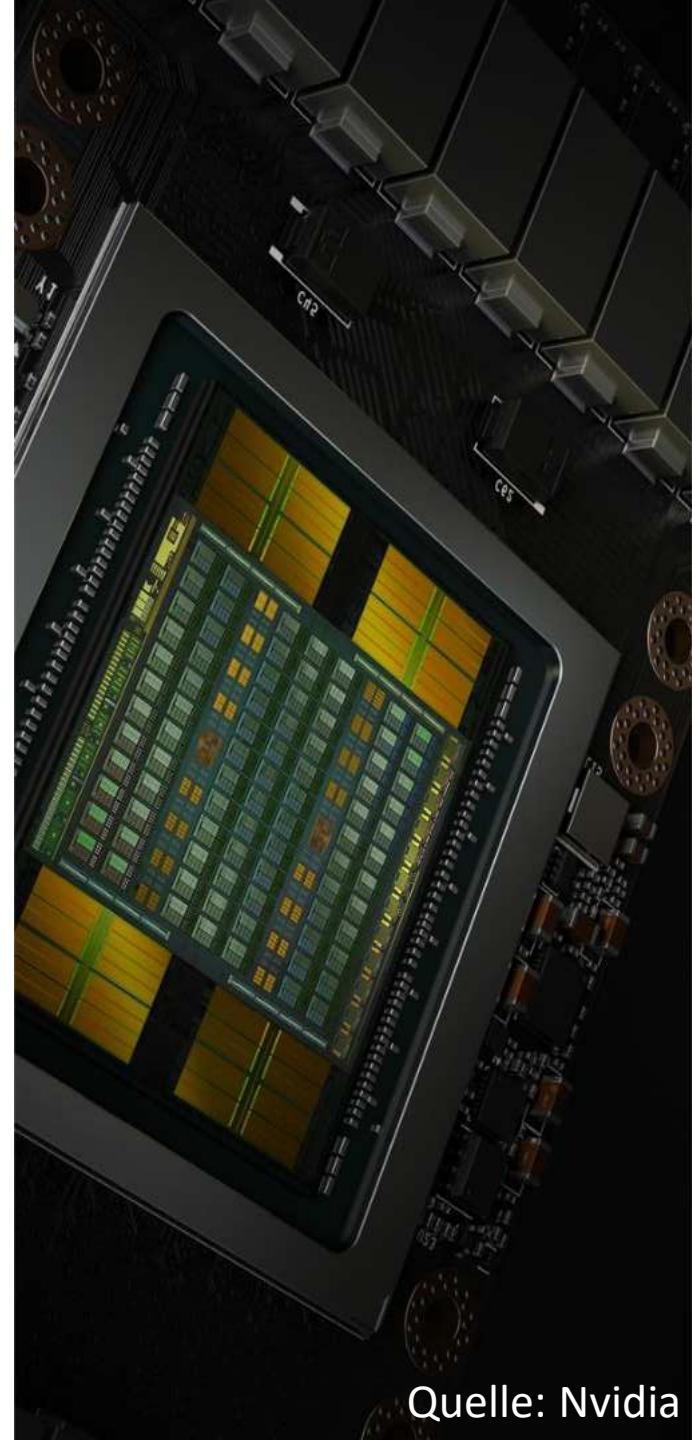
---

- Parallelle Methoden für Arrays:
  - *Sortieren, Werte setzen, etc.*
  - *implementiert in `java.util.Arrays`*
- Java 1.8 hat einen Default Fork/Join-Pool, den die obigen parallelen Methoden verwenden
  - Zugriff: `ForkJoinPool.commonPool()`
- Streams: flexible Berechnungen auf großen Datenmengen
  - *Unterstützt Parallelisierung: Klasse `ParallelStream`*
  - *Achtung: nicht alles dort wird wirklich parallel berechnet!*

# GPU Programmierung

Prof. Dr. rer. nat. Tobias Lauer  
Dr.-Ing. Alexander Vondrouš

Hochschule Offenburg



Quelle: Nvidia

# Motivation

---

## Mathematik

- Numerik

## Computational Finance

- Simulationen (Monte Carlo)
- Risikoanalyse
- Derivatbewertung

## Signalverarbeitung

- Videoanalyse und -bearbeitung
- Sensordaten

## Deep Learning

- Training Neuronaler Netze

## Physik

- Teilchensimulationen

## Biologie, Bioinformatik

- Sequence alignment
- Proteinfaltung

## Kryptologie

- Passwortsicherheit

## Meteorologie

- Wetter- und Klimasimulationen

## Mining

- Bitcoin etc.

uvm.

# GPGPU Motivation

---

## General Purpose Computation on Graphics Processing Unit

- GPUs bieten eine sehr hohe Rechenleistung pro Euro (u.a. auch durch niedrigere0n Energieverbrauch pro FLOP/s).
- GPUs können MAC (Multiply-Accumulate) Operationen in einem Takt Ausführen (Number Cruncher).
- Ausgefeilte SIMD Hardware für datenparallele Anwendungen
- GPUs sind zugänglich und in vielen Workstations und Laptops eingebaut. Jeder Fachhändler hat sie im Sortiment.
- Viele rechenintensive Anwendungen können mit GPUs signifikant beschleunigt werden.

# Rückblick

---

**Shaderprogrammierung** mit OpenGL 1992, Glide 1996 oder anderen Shader Programmiersprachen bzw. Spezifikationen sind umständlich.

2007 wurde **CUDA** (Compute Unified Device Architecture) von Nvidia für ausschließlich Nvidia GPUs veröffentlicht und erleichtert die Programmierung wesentlich.

2009 wurde **OpenCL** (Khronos Group) als Standard veröffentlicht, somit kann auf Nicht-Nvidia Grafikkarten programmiert werden.

2011 wurde **OpenACC** (PGI, Cray, Nvidia, CAPS) veröffentlicht, damit CPU und GPU Programmierung portierbar und in heterogenen Umfeld (diverse Hersteller: CPU, CPU-GPU, GPU) genutzt werden kann.

# GPUs Heute

---

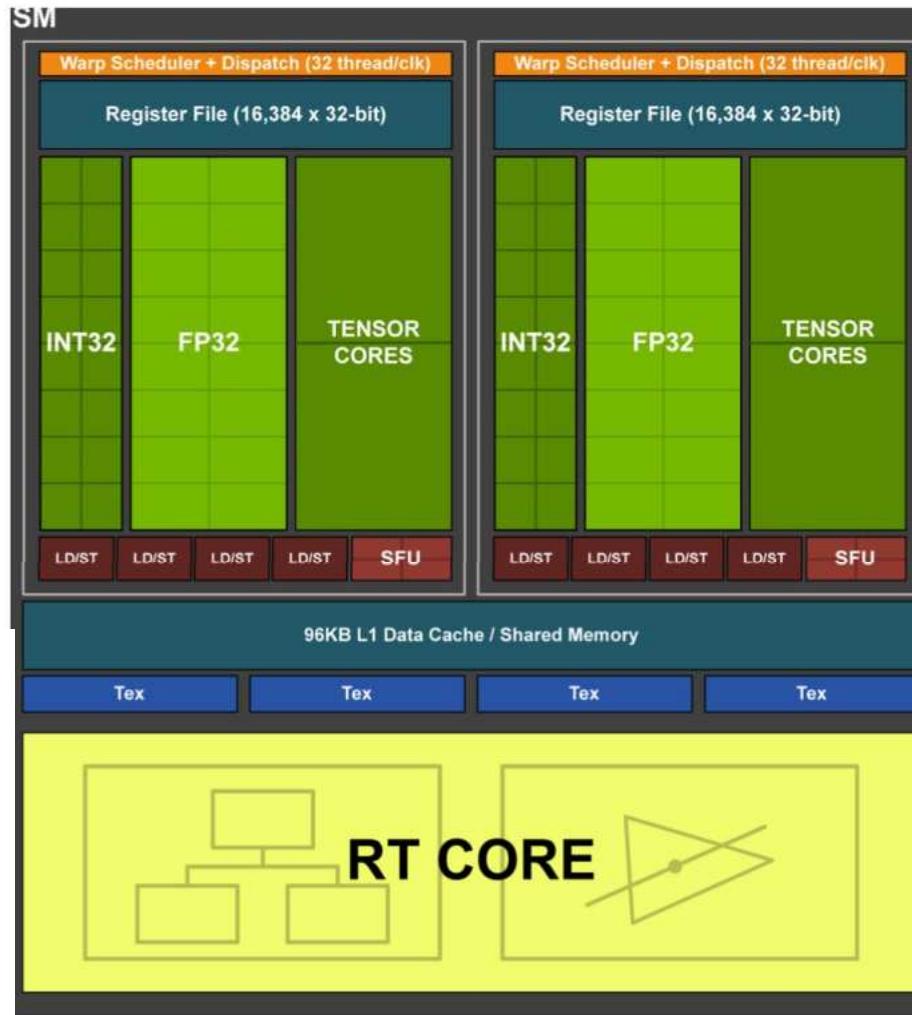
- „General-Purpose Computation on Graphics Processing Units“
  - 2009
    - Erste GPU Technology Conference von NVIDIA
  - 2010
    - Schnellster Supercomputer „Tianhe-1A“ mit 7.168 GPUs (NVIDIA Tesla M2050)
  - 2012
    - Schnellster Supercomputer „Titan“ mit 18.688 GPUs (NVIDIA Tesla K20X)
  - 2014
    - GPUs in 17 der Top-100 Supercomputer
  - 2019
    - Summit Supercomputer mit 27 648 GPUs und 9 216 CPUs

# GPGPU und Algorithmitk

---

- Erstmalige Möglichkeit, massiv (daten-)parallele Algorithmen auf kostengünstiger Hardware zu realisieren
  - Theorie der parallelen Algorithmen reicht zurück bis in die 1980er Jahre und weiter
  - Damals nur wenige Spezialrechner
- Skalierbarkeit der Prozessorzahl mit der Datengröße prinzipiell möglich
  - z.B. durch Multi-GPU
- Relativ einfache Implementierung
  - Vor allen durch CUDA und OpenCL

# GPU Aufbau



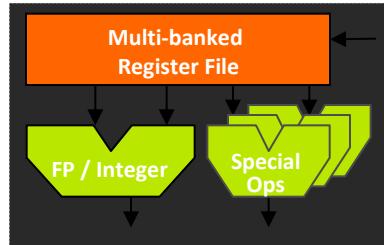
# GPU Aufbau



# Hardware: Überblick (GTX 280)

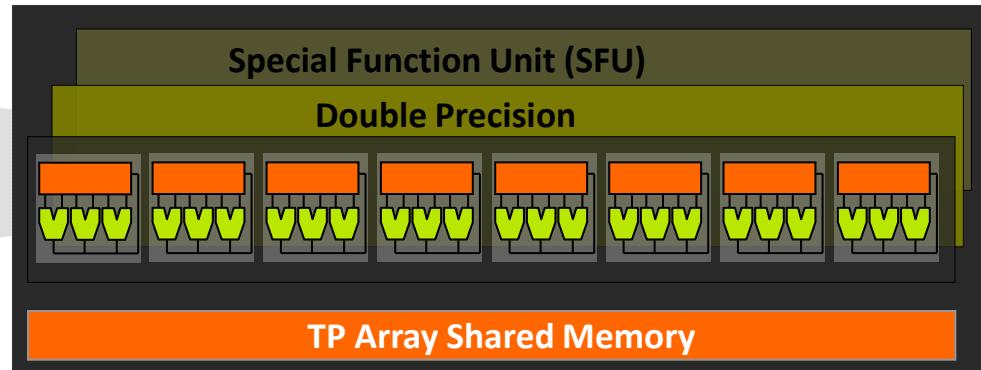
Core

Streaming Processor (SP)  
Thread Processor (TP)

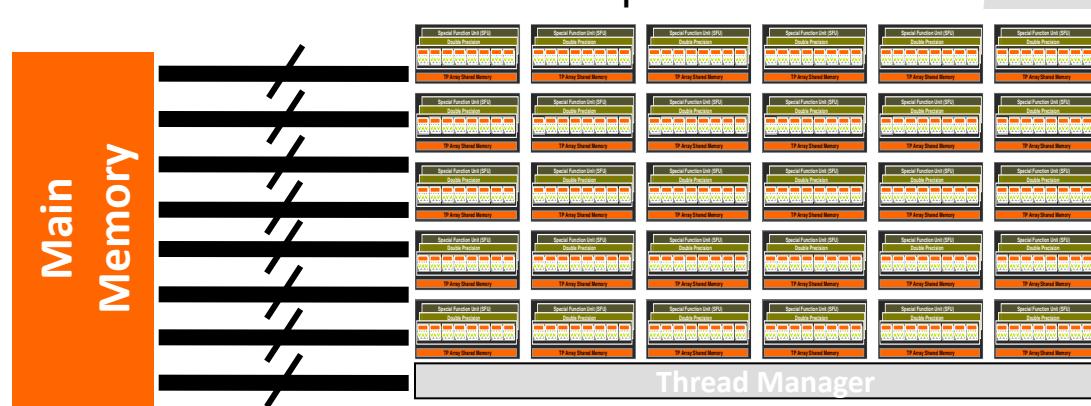


Processor

Streaming Multiprocessor (SM)



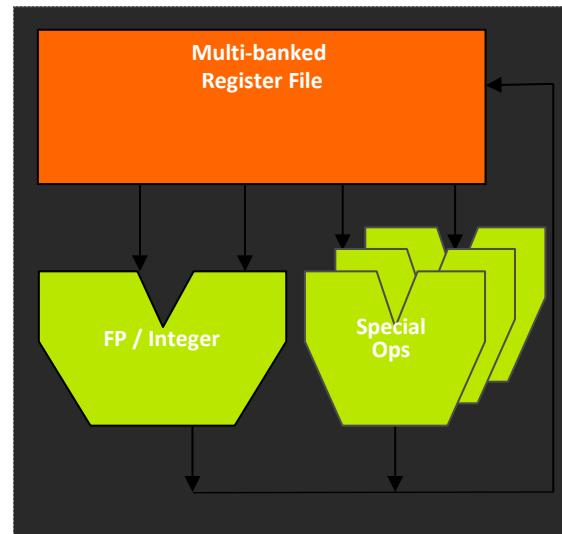
Chip



30 SMs pro chip

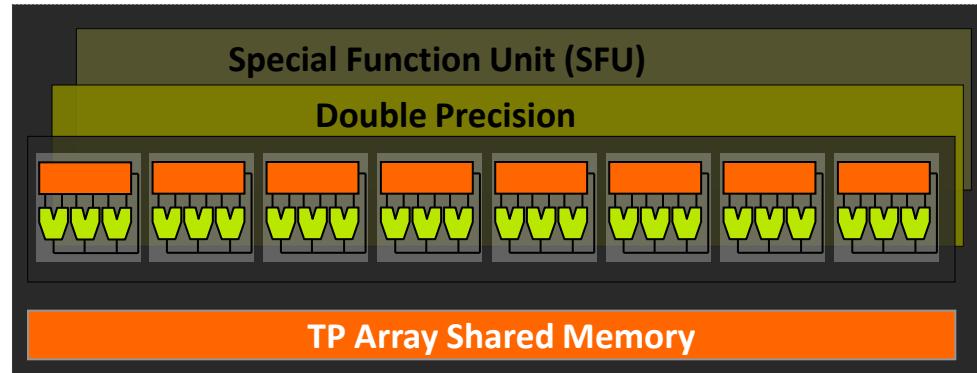
# Threadprozessor (CUDA Core)

- Floating point / Integer Einheit
- Move, compare, logic, branch
- Lokale Register
- „Abgespeckter“ Prozessorkern bzw. hochgezüchtete ALU



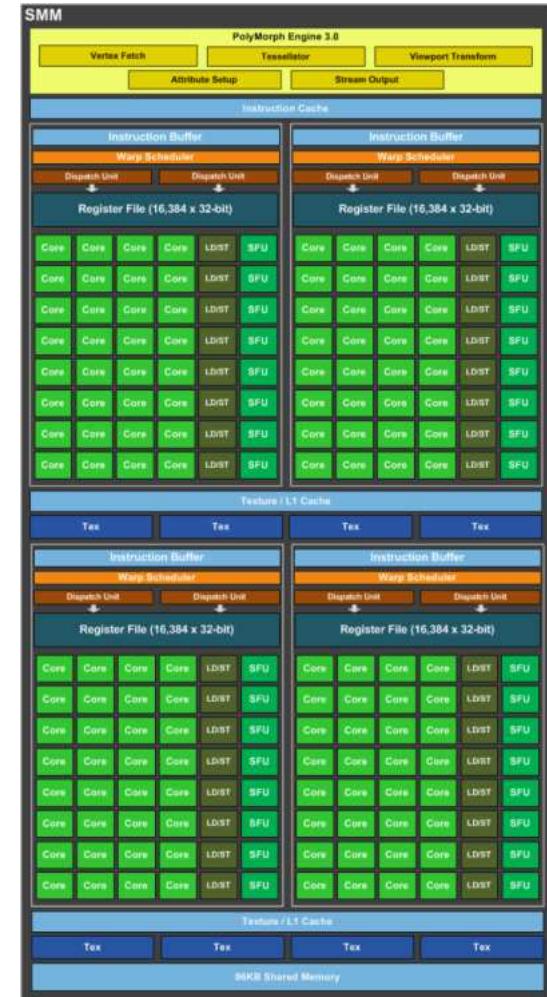
# Multiprozessor

- 8 / 32 / 192 / 128 / 64 Threadprozessoren (CUDA Cores)
- Double-precision Einheit (IEEE-754-konform)
- 16 KB / 64 KB Shared Memory
- Gemeinsame Instruktionseinheit und Instruktionen-Cache
- Simultane Verwaltung einer großen Anzahl Threads
  - Hunderte Threads gleichzeitig
  - Hardware-Scheduler mit Kontextswitching fast ohne Overhead
- NVIDIA GTX 280: 30 Multiprozessoren (à 8 CUDA Cores)
- NVIDIA Tesla P100: 56 Multiprozessoren (à 64 CUDA Cores)



# Multiprozessor – Maxwell-Generation

- Bis zu 16 Multiprozessoren pro GPU (z.B. GeForce GTX 980)
- Pro Multiprozessor:
  - 128 Threadprozessoren (CUDA Cores)
  - 64-96 KB Shared Memory
  - 64 KB Register file
  - Bis zu 64 aktive Warps (à 32 Threads)  
→ 32768 aktive Threads



# Multiprozessor – Pascal-Generation

- Bis zu 60 Multiprozessoren pro GPU  
(z.B. GeForce GTX 1080)
- Pro Multiprozessor:
  - 64 Threadprozessoren (CUDA Cores)
  - 64 KB Shared Memory
  - 64 KB Register file
  - Bis zu 64 aktive Warps (à 32 Threads),  
→ 122880 aktive Threads



# Multiprozessor – Turing-Generation

- Bis zu 72 Multiprozessoren pro GPU (z.B. Quadro RTX 6000)
- Pro Multiprozessor:
  - 64 Threadprozessoren (CUDA Cores)
  - 64 KB Shared Memory
  - 64 KB Register File



# Prozessorarray

---

- Skalierbar by Design
  - Multiprozessoren bilden **skalierbares Prozessorarray**
  - Variierende Anzahl von Multiprozessoren je nach GPU  
**(automatische skalierende Ausführung)**
- Architektur hat Auswirkung auf Programmierung
  - Jeder Multiprozessor für sich ist eine **SIMD-ähnliche Einheit**

# SIMD – Single instruction, multiple data

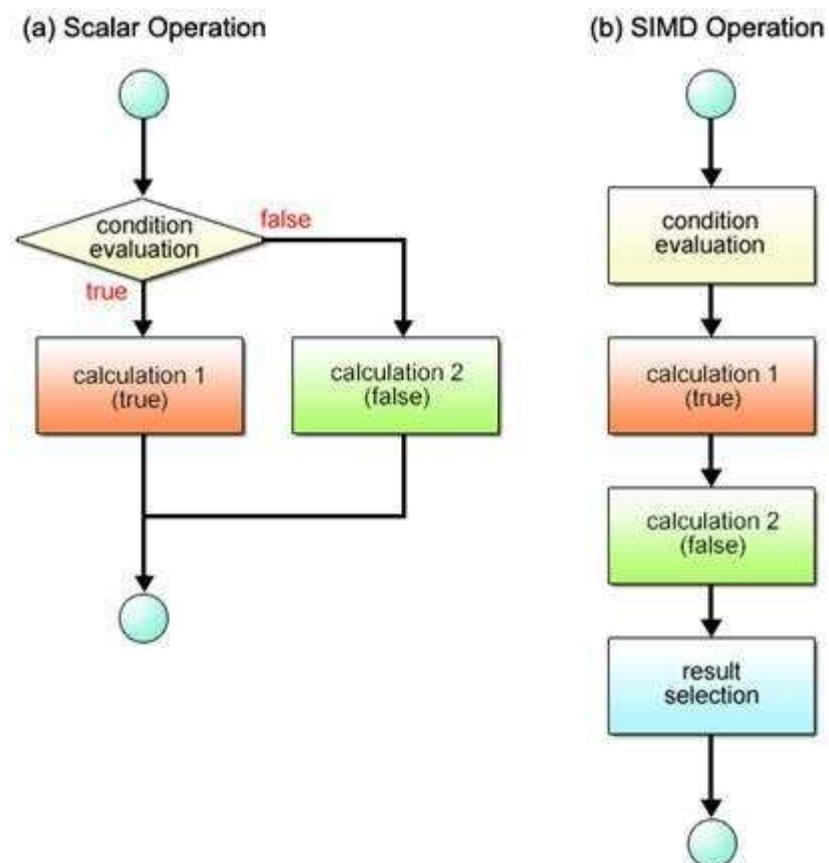
---

- Alle Prozessor führen in jedem Schritt **dieselbe Instruktion** aus, aber auf **unterschiedlichen Daten**
- vgl. MIMD (multiple instruction, multiple data)
  - Jeder Prozessor kann ein beliebiges Programm ausführen
  - Standard Multicore-Architektur (Dualcore, Quadcore, ...)
- Wenn alle immer dieselbe Instruktion ausführen, wie kann man dann Verzweigungen (if ... else) realisieren?

# Conditionals/branches in SIMD

## Elimination von Branching

- Parallele Evaluierung des Prädikats für alle Elemente
- Parallele Berechnung des TRUE-Zweigs für **alle** Elemente
- Parallele Berechnung des FALSE-Zweigs für **alle** Elemente
- Parallele Selektion des korrekten Werts für alle Elemente (**wie?**)



# Speichersystem

---

- Bis zu 24 GB Speicher unterstützt  
(z.B. Turing Quadro P6000)
- Generisches load/store Modell  
([Concurrent Read](#), [Concurrent Write](#))
  - ACHTUNG: [arbitrary CRCW](#)
  - Jeder Prozessor kann in jede Speicherstelle schreiben
  - Konflikte ([race condition](#)) möglich
  - → Programmierer\*in ist in der Verantwortung!

# Speichertypen

---

- **Global memory**

- Schreiben/Lesen
- Groß: hunderte MB bis 32 GB/GPU
- Langsam (600+ Zyklen)

- **Texture memory**

- Physisch dasselbe wie Global memory
- Read-only
- Cached für Streaming (2D Nachbarschaften)

# Speichertypen

---

- Constant memory
  - Read-only
  - 64kB pro Chip
  - Sehr schnell (1-4 Zyklen)
- Shared memory (z.T. geteilt mit L1 Cache)
  - Read/write
  - 16kB - 64kB pro Multiprozessor
  - Sehr schnell, sofern DRAM Bankkonflikte vermieden werden
- Register
  - Read/write
  - 16kB-64kB pro Multiprozessor; max. 255 Register pro Thread
  - Schnellster Speicher

# Speichertypen

---

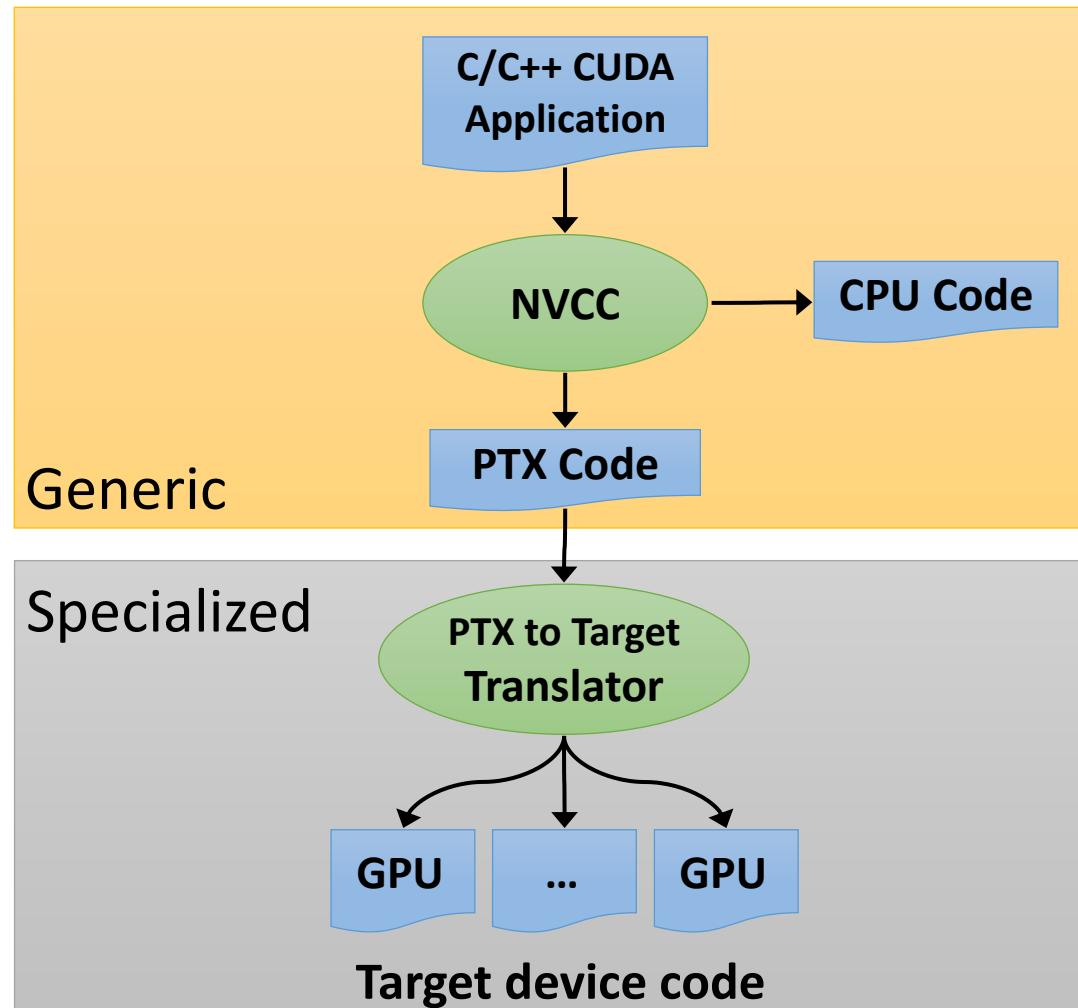
- **Performanz** von Anwendungen hängt entscheidend von der effizienten Verwendung der Speicherhierarchie ab!
- Zum Beispiel
  - Shared memory als Cache für Global Memory
  - Constant memory zur Übergabe von größeren Parameter-Sets
  - Effiziente Registerverwendung
  - ...

# Programmiermodell: CUDA (Compute-Unified Device Architecture)

---

- Skalierung auf
  - Hunderte bis tausende von Cores
  - Zigtausende bis Millionen von Threads
- Programmierer\*in soll sich auf parallele Algorithmen konzentrieren können
  - Keine spezielle parallele Programmiersprache
  - C for CUDA plus Runtime-API
- Unterstützung heterogener Systeme (d.h. CPU+GPU)
  - CPU & GPU sind separate Geräte mit separaten DRAMs
    - Heute transparent (unified address space, unified memory)
  - CPU = “Host”, GPU = “Device”

# Programmiermodell: CUDA



# Parallele Abstraktionen in CUDA

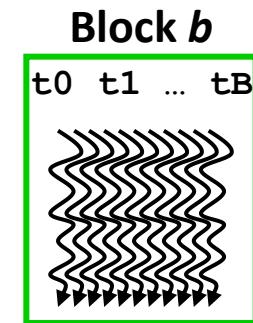
---

- Hierarchie von nebenläufigen Threads
- Leichtgewichtige Synchronisationsprimitive
- Shared-Memory-Modell für kooperierende Threads

# Hierarchie nebenläufiger Threads

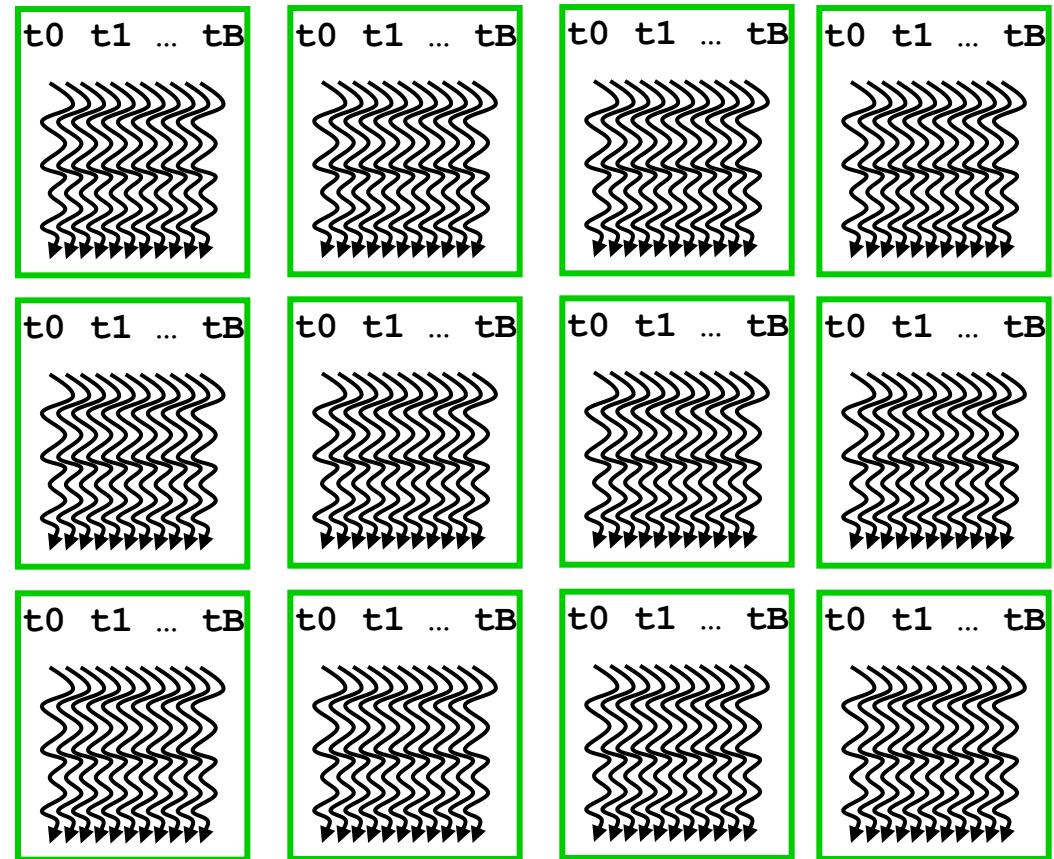
- Parallele **Kernels** mit vielen Threads
  - Alle Threads führen dasselbe sequenzielle Programm aus
- Threads sind in **Thread-Blocks** gruppiert
  - Blocks sind 1-, 2- oder 3-dimensional angeordnet
  - Threads im selben Block können kooperieren
- Threads/Blocks haben eindeutige IDs
- Thread-Blocks sind in einem **Grid** gruppiert
  - Grid is 1-, 2- oder 3-dimensional angeordnet

Thread *t*



# Grid-Struktur

- Thread-Blocks sind als 1- oder 2-dimensionales Grid angeordnet
- Dadurch einfache Anpassung der Grid-Struktur an die Problem-Struktur
- Jeder Thread ist durch seine Position im Block und der Position des Blocks im Grid eindeutig identifizierbar



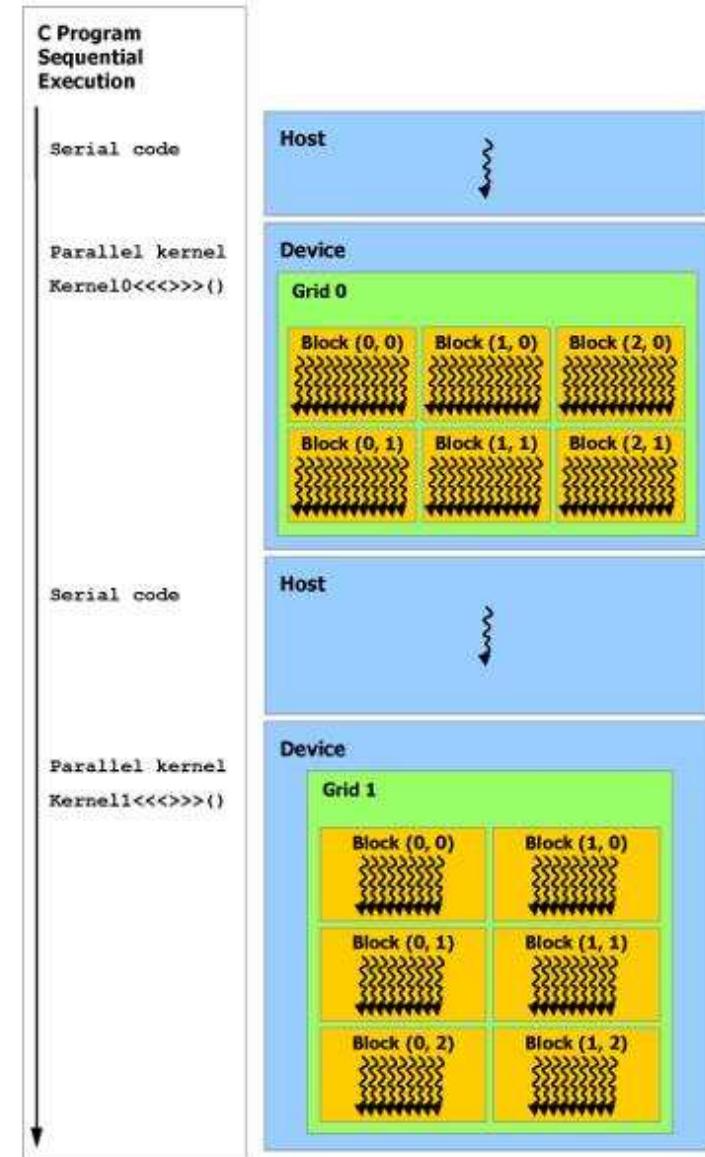
# CUDA-Programme

---

- GPU ist Coprozessor
  - Host-Code und Device-Code
  - Applikationsgesteuerte Aufrufe von parallelen Funktionen
- Kein „strenges“ SIMD, sondern
  - SPMD – Single Program, Multiple Data
    - Zwischen verschiedenen Warps  
(Warp = Thread-Bündel, das als Einheit gescheduled wird)
    - Dasselbe Programm (aber nicht unbedingt dieselbe Instruktion) wird gleichzeitig auf allen Daten ausgeführt
  - SIMT – Single Instruction, Multiple Thread
    - Innerhalb eines Warps
    - Jeder Thread hat eigene Register

# Typische CUDA-Progammstruktur

- Anwendung wird in sequenzielle und parallel berechenbare Teile/Tasks aufgespalten
- Für jeden parallelisierbaren Teil wird
  - Speicher auf GPU(s) zugewiesen
  - Nötige Daten vom RAM in den GPU-Speicher kopiert
  - CUDA-Kernel(s) gestartet
  - Ergebnis(se) zurück zum Host kopiert
  - Speicher freigegeben



# Beispiel: Vektoraddition

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Device Code

# Beispiel: Vektoraddition

```
// Compute vector sum C = A+B  
// Each thread performs one pair-wise addition  
__global__ void vecAdd(float* A, float* B, float* C)  
{  
    int i = threadIdx.x + blockDim.x * blockIdx.x;  
    C[i] = A[i] + B[i];  
}
```

Host Code

```
int main()  
{  
    // Run N/256 blocks of 256 threads each  
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);  
}
```

# Synchronisation

---

- Threads innerhalb eines Blocks können mit Barrieren synchronisiert werden
  - ... Step 1 ...  
`__syncthreads();`  
... Step 2 ...
- Koordination mithilfe atomarer Speicherzugriffsoperationen
  - z.B. Inkrementieren eines gemeinsamen Pointers mit `atomicInc()`
- Implizite Barriere zwischen abhängigen Kernels
  - `vec_minus<<<nblocks, blksize>>>(a, b, c);`  
`vec_dot<<<nblocks, blksize>>>(c, c);`
- ACHTUNG: Unterschiede zwischen Architekturen (Compute Capability)!

# Compute Capabilities

---

- CC1.1 atomare int32-Operationen im Global Memory
- CC1.2 atomare int32-Operationen im Shared Memory
  - atomare int64-Operationen im Global Memory
- CC1.3 Gleitkommaarithmetik mit doppelter Präzision
- CC2.x atomare int64-Operationen im Shared Memory
  - atomicAdd auf 32-bit floats
  - neue Synchronisations-Mechanismen
- CC3.5 Dynamic parallelism
  - shuffle-Instruktion (Threads tauschen Daten)
- ...
- CC7.5 Tensor Cores für Training von DNN

# Was ist ein Thread?

---

- Unabhängiger Ausführungsstrang
  - Hat eigenen Program Counter, Variablen (Register), Prozessor Status, etc.
  - Kein Einfluss auf Thread-Scheduling
- CUDA Threads können **physische** Threads sein
  - „regulär“ auf NVIDIA GPUs
- CUDA Threads können **virtuelle** Threads sein
  - z.B. 1 Block = 1 physischer Thread auf Multicore-CPUs (wie in MCUDA)

# Was ist ein Thread-Block?

---

- Thread-Block = **virtualisierter Multiprozessor**
  - Frei wählbare “Prozessorgröße” für Datenmenge
  - Kann für jeden Kernel-Start angepasst werden
- Thread-Block = ein (daten)**paralleler Task**
  - Alle Blocks in einem Kernel haben denselben Einstiegspunkt
  - Aber können beliebigen Code ausführen
- Thread-Blocks eines Kernels müssen **unabhängige Tasks** sein
  - Blockausführungen können beliebig „interleaved“ sein

# Blocks müssen unabhängig sein

---

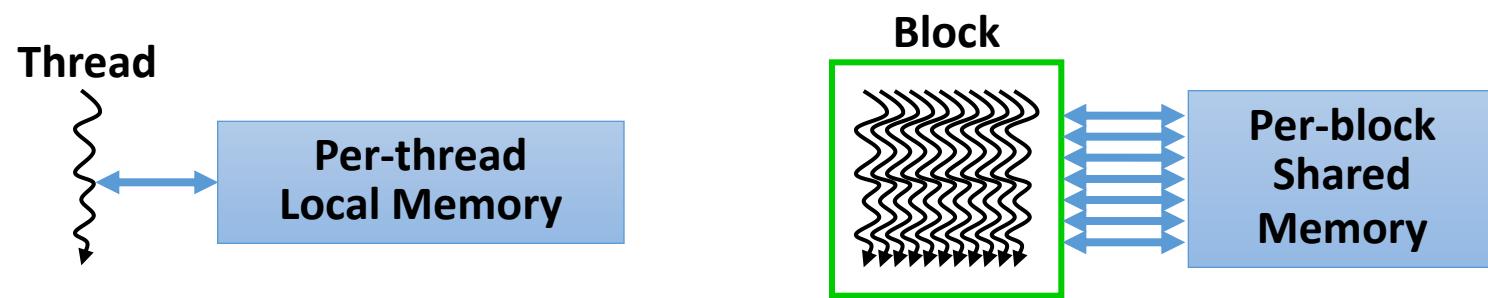
- Beliebiges Interleaving von Blocks sollte zu korrektem Ergebnis führen
  - Laufen ohne vorher bestimmte Reihenfolge
  - Können nebenläufig ODER sequenziell ausgeführt werden
- Blocks können koordinieren, aber **nicht synchronisieren**
  - Shared Queue pointer: **OK**
  - Shared Lock: **SCHLECHT** ... Gefahr von Deadlocks
- Unabhängigkeitsforderung garantiert **Skalierbarkeit**
  - Und macht die Hardwarerealisierung handhabbar

# Ebenen der Parallelität

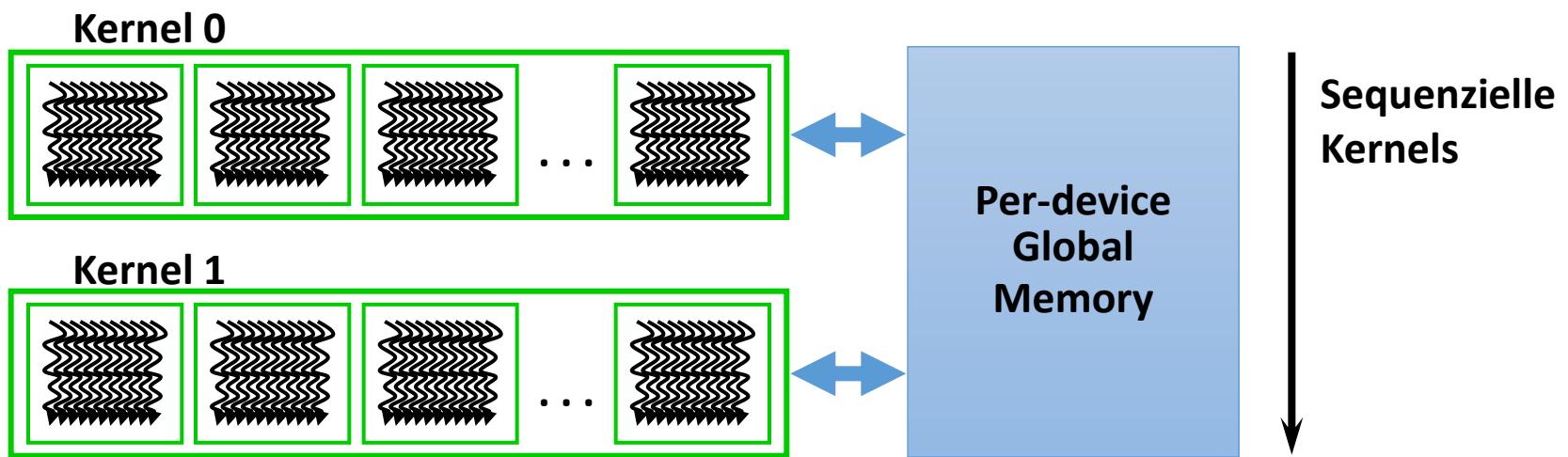
---

- Threadparallelität
  - Jeder Thread ist ein unabhängiger Ausführungsstrang
- Datenparallelität
  - Zwischen Threads in einem Block
  - Zwischen Blocks in einem Kernel
- Taskparallelität
  - Verschiedene Blocks sind unabhängig
  - Kernels sind unabhängig

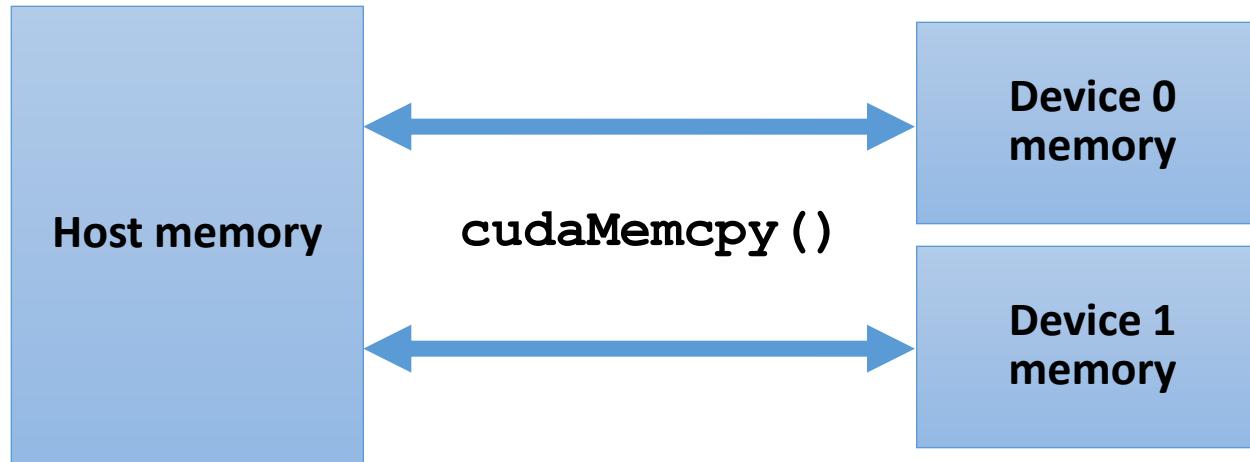
# Speichermodell



# Speichermodell



# Speichermodell



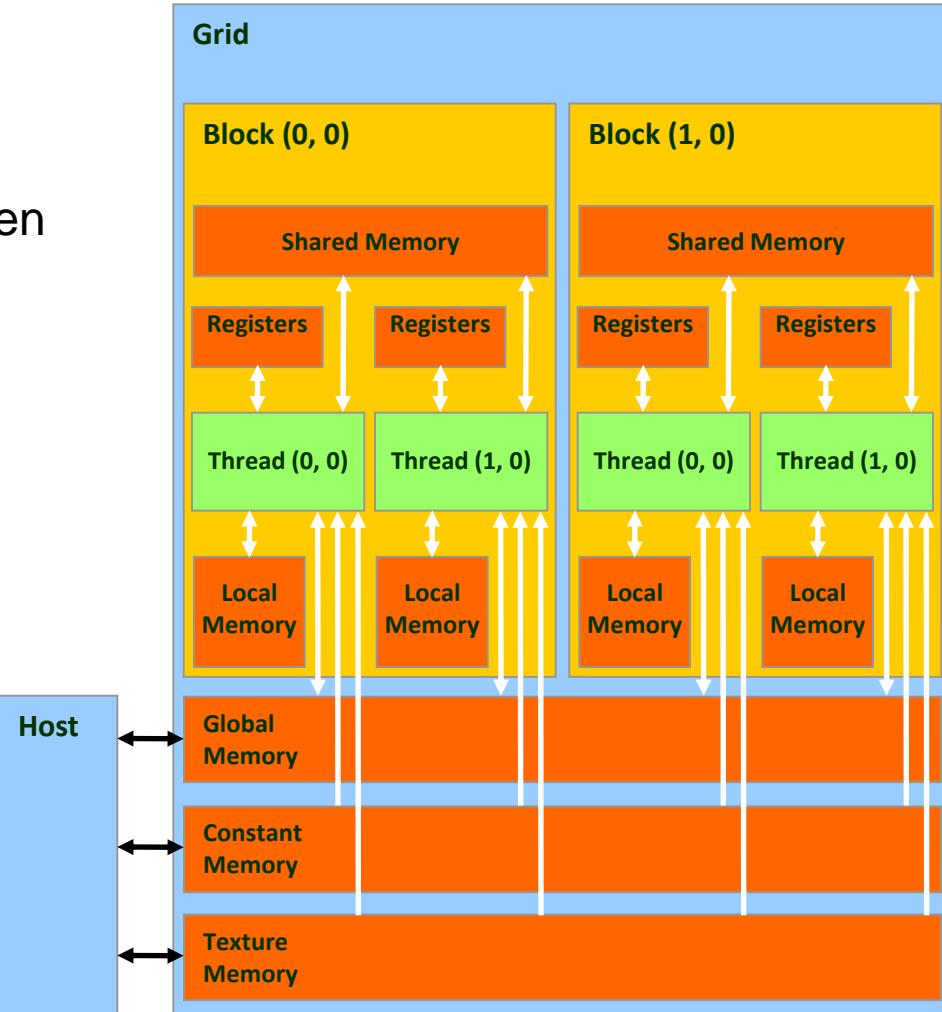
# Speichermodell

## Jeder Thread kann

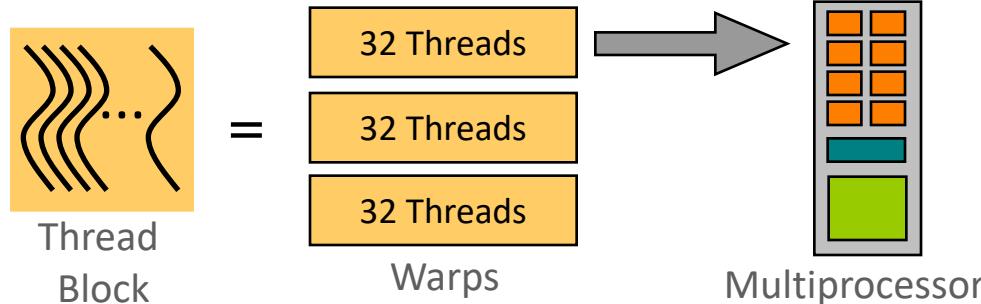
- seine Register lesen/schreiben
- seinen Local memory lesen/schreiben
- im Shared memory seines Blocks lesen/schreiben
- Global memory lesen/schreiben
- Constant memory nur lesen
- Texture memory nur lesen

## Der Host (CPU) kann

- Global memory lesen/schreiben
- Constant memory lesen/schreiben
- Texture memory lesen/schreiben

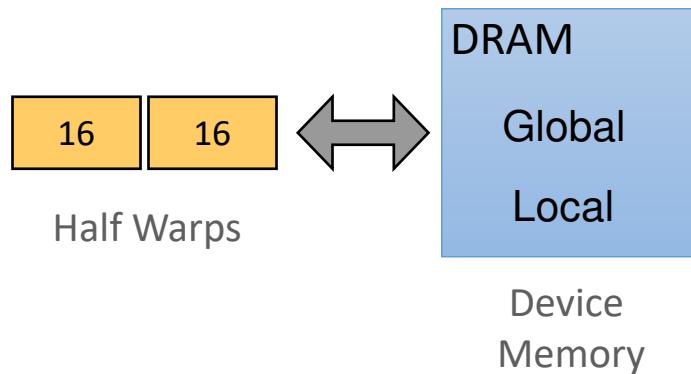


# Warps und Half-Warps



Ein Thread-Block besteht aus **Warps** mit je 32 Threads.

Ein Warp wird physisch parallel auf demselben Multiprozessor ausgeführt.



Ein Half-Warp aus 16 Threads kann Zugriffe auf den Global Memory als einzelne Transaktion ausführen (**coalescing**)

# C for CUDA: Minimale Erweiterungen

- Deklarationen (geben an, wo die Dinge sind)

```
__global__ void KernelFunc(...); // Kernel aufrufbar vom Host
__device__ void DeviceFunc(...); // Funktion aufrufbar auf dem Device
__device__ int GlobalVar;      // Variable im Device memory
__shared__ int SharedVar;     // Variable im Block-internen Shared memory
```

- Erweiterte Syntax für den Aufruf paralleler Kernels

```
KernelFunc<<<500, 128>>>(...); // 500 Blocks, 128 Threads pro Block
KernelFunc<<<500, 128, 1024>>>(...); // ... 1024B Shared memory pro Block
```

- Spezielle Variablen zur Thread-Identifizierung in Kernels

```
dim3 threadIdx; dim3 blockIdx; dim3 blockDim;
```

- Intrinsics für spezifische Operations im Kernel-Code

```
__syncthreads(); // Synchronisationsbarriere
```

# Runtime Support

---

- Explizite Speicherallokation, gibt Pointer auf GPU-Memory zurück  
`cudaMalloc()`, `cudaFree()`
- Explizites Kopieren für Host-Device, Device-Device  
(seit CUDA 4.0 auch zwischen verschiedenen Devices)  
`cudaMemcpy()`, `cudaMemcpyPeer()`, ...
- Interoperabilität mit OpenGL & DirectX  
`cudaGLMapBufferObject()`, `cudaD3D9MapVertexBuffer()`, ...

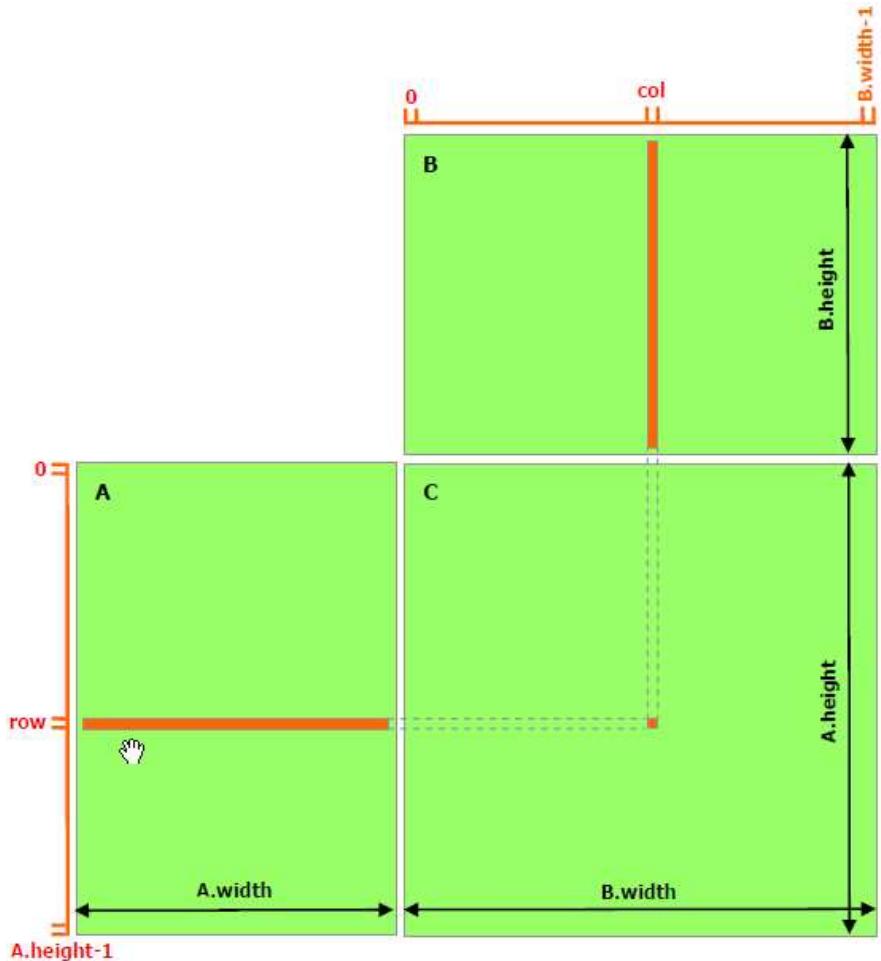
# Zusammenfassung

---

- CUDA = C + einfache Erweiterungen
  - Einfacher Start beim Schreiben grundlegender paralleler Programme
- Drei wichtige Abstraktionen:
  1. Hierarchie paralleler Threads
  2. Korrespondierende Ebenen der Synchronisation
  3. Korrespondierende Speicherorte
- Unterstützt massive Parallelität von Manycore-GPUs

# Beispiel: Matrixmultiplikation

- Zwei Matrizen  $A$  und  $B$  werden multipliziert:
  - Das Vektor-Skalarprodukt der  $i$ -ten Zeile von  $A$  mit der  $j$ -ten Spalte von  $B$  ergibt den Eintrag  $C_{i,j}$  in der Ergebnismatrix  $C$
- $A$  und  $B$  müssen richtig dimensioniert sein, um multipliziert werden zu können.



# Beispiel: Matrixmultiplikation ( $n \times n$ )

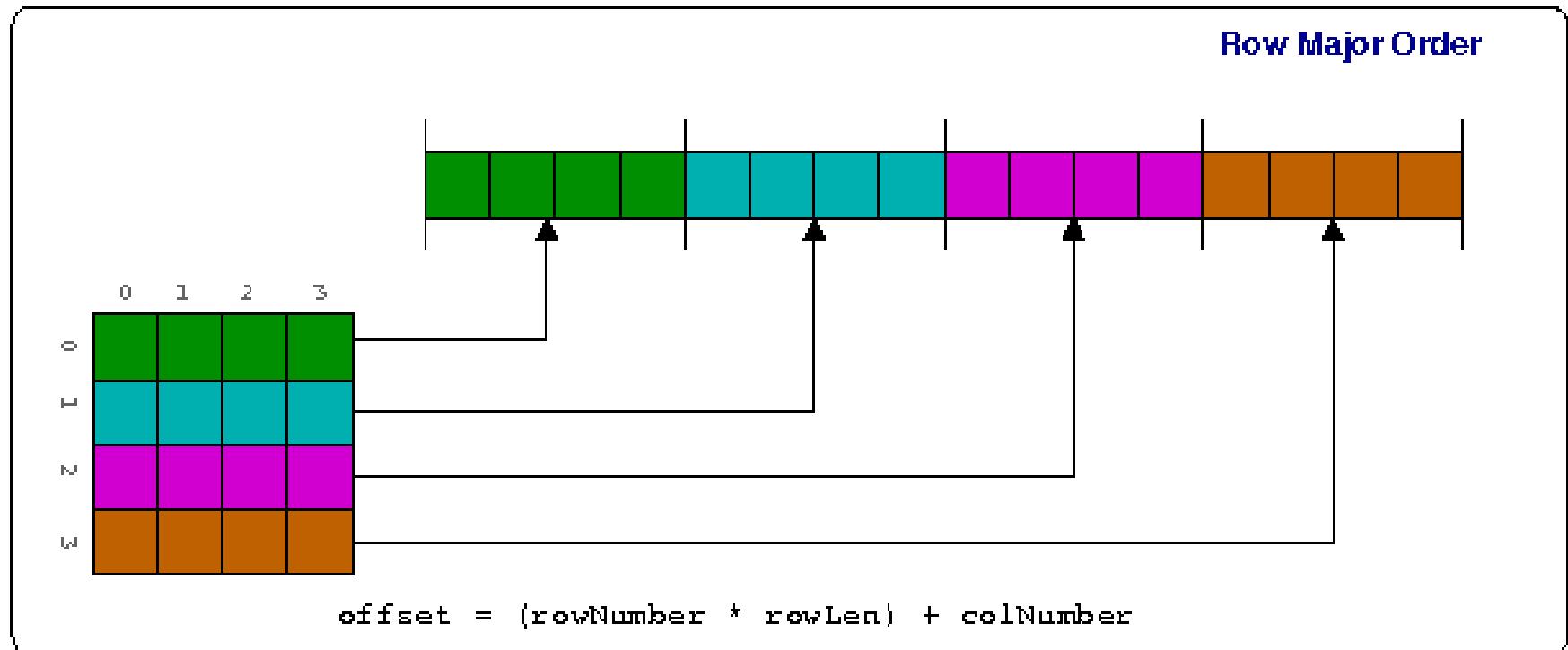
---

- Zwei ( $n \times n$ )-Matrizen  $A$  und  $B$  werden multipliziert:
  - Das Vektor-Skalarprodukt der  $i$ -ten Zeile von  $A$  mit der  $j$ -ten Spalte von  $B$  ergibt den Eintrag  $C_{i,j}$  in der Ergebnismatrix  $C$
  - Vektor-Skalarprodukt:  $a \cdot b = a_0 \cdot b_0 + a_1 \cdot b_1 + \dots + a_{n-1} \cdot b_{n-1}$
- Sequenzieller Algorithmus

```
for (i = Zeilen von A)
    for (j = Spalten von B)
        for (k = Wert in Zeile i, Spalte j)
            C[i,j] += a[i, k] * b[k, j]
```

# Repräsentation

Matrix als **eindimensionales Array**  
in **row-major** Darstellung



# Repräsentation

Explizite **row-major** Darstellung:

Aus **M[i][j]** wird **M[i\*width + j]**

Sequenzieller Algorithmus:

```
for (int i=0; i<width; i++) {  
    for (int j=0; j<width; j++) {  
        float scalar = 0;  
        for (int k=0; k<width; k++) {  
            float a = A[i*width + k];  
            float b = B[k*width + j];  
            scalar += a * b;  
        }  
        C[i*width + j] = scalar;  
    }  
}
```

# Matrixmultiplikation

---

- Beobachtungen:
  - Jeder Eintrag der Ergebnis-Matrix kann unabhängig von den anderen berechnet werden
  - Reihenfolge aller Schleifen ist irrelevant
- Idee:
  - Ein Thread für jeden Eintrag ( $i, j$ ) der Ergebnis-Matrix
  - „**Embarrassingly parallel**“ (keinerlei Abhängigkeiten)
- Theoretisch könnte auch die **innerste Schleife (Skalarprodukt)** parallelisiert werden

# Ein Thread pro Matrixeintrag

---

- Mache Thread-Block 2-dimensional, um die Struktur der Matrix abzubilden.
  - Thread-Position im Block: (x, y)
- Thread an Position (*i*, *j*) im Block berechnet Eintrag (*i*, *j*) der Ergebnismatrix:

```
int i = ThreadId.x;
int j = ThreadId.y;
float scalar = 0;
for (k=0; k<width; k++) {
    float a = A[i*width + k];
    float b = B[k*width + j];
    scalar += a * b;
}
C[i*width + j] = scalar;
```

# Umsetzung in CUDA C

---

- **Initialisierung**
  - Speicherallokation auf Host-Seite, Einlesen des Inputs
  - Speicherallokation auf Device-Seite: `cudaMalloc`
  - Kopieren des Inputs vom Host zum Device: `cudaMemcpy`
- GPU-Kernel starten
  - `MatrixMulKernel<<<blocksize, threads>>>( . . . )`
- Abschluss
  - Kopieren des Outputs vom Device zum Host: `cudaMemcpy`
  - Speicher freigeben: `cudaFree`

# Initialisierung (Host)

---

```
// Speicherallokation für Matrizen M und N (Host)
unsigned int size = width * width * sizeof(float);
float* hostM = (float*) malloc(size);
float* hostN = (float*) malloc(size);

// Matrizen mit Werten befüllen
...

// Speicherallokation für Ergebnismatrix P (Host)
float* hostP = (float*) malloc(size);
```

# Initialisierung (Device)

---

```
// Speicherallokation für Matrix M (Device)  
float* deviceM;  
cudaMalloc((void**)deviceM, size);
```

Erster Parameter von *cudaMalloc*:

Adresse einer Pointers, der nach der Allokation auf das allokierte Objekt verweist

Cast auf `(void**)` wird empfohlen, da ein generischer Pointer erwartet wird (und keiner auf einen bestimmten Typ)

Zweiter Parameter:

Anzahl zu allozierender Bytes

# Kopieren vom Host zum Device

---

```
// Speicherallokation für Matrix M (Device)
float* deviceM;
cudaMalloc((void**)&deviceM, size);

// Kopiere Matrix M vom Host zum Device
cudaMemcpy(deviceM, hostM, size, cudaMemcpyHostToDevice);
```

Parameter von *cudaMemcpy* :

- Pointer auf Ziel-Location (Device)
- Pointer auf Quell-Objekt (Host)
- Anzahl von Bytes, die kopiert werden sollen
- Involvierte Speicherorte (Quelle, Ziel)

# Umsetzung in CUDA C

---

- Initialisierung
  - Speicherallokation auf Host-Seite, Einlesen des Inputs
  - Speicherallokation auf Device-Seite: `cudaMalloc`
  - Kopieren des Inputs vom Host zum Device: `cudaMemcpy`
- GPU-Kernel starten
  - `MatrixMulKernel<<<grid, block>>>( . . . )`
- Abschluss
  - Kopieren des Outputs vom Device zum Host: `cudaMemcpy`
  - Speicher freigeben: `cudaFree`

# Kernel starten

---

```
// CUDA-Ausführungskonfiguration definieren  
dim3 dimBlock(width, width);  
dim3 dimGrid(1, 1); // Achtung, unrealistisch!  
  
// Kernel aufrufen  
MatrixMulKernel<<<dimGrid, dimBlock>>>(deviceM, deviceN,  
deviceP, width);
```

Kernelaufruf erfolgt durch:

Name der Kernelfunktion

<<<CUDA-Konfiguration>>>

Parameter der Kernelfunktion

**dim3**: Struct zur Darstellung der Dimensionen von Blocks und Grids.

# Konfiguration

---

```
// CUDA-Ausführungskonfiguration definieren  
dim3 dimBlock(width, width);  
dim3 dimGrid(1, 1);
```

Beschränkungen für Block- und Grid-Konfiguration (CC 1.x):

- Block: max. Dimensionen 512 x 512 x 64
  - und Gesamtgröße  $\leq$  512
  - und Gesamtgröße Vielfaches von 32 (CC 1.x)
  - bzw. 1024 x 1024 x 64
  - und Gesamtgröße  $<$  1024 (CC 2.0+)
- Grid: max. Dimensionen 65535 x 65535 x 1 (CC 1.x)
  - bzw. 65535 x 65535 x 65535 (CC 2.x)
  - bzw.  $2^{31}-1$  x 65535 x 65535 (CC 3.0+)

Quelle: <http://en.wikipedia.org/wiki/CUDA>

# Kernel-Code

---

```
// Kernel-Funktion, wird von jedem Thread ausgeführt
__global__ void MatrixMulKernel(float* A,
                                float* B,
                                float* C,
                                int width)
{
    int i = ThreadIdx.x; // x-Koordinate des Threads im
    Block
    int j = ThreadIdx.y; // y-Koordinate des Threads im
    Block
    float scalar = 0;
    for (k=0; k<width; k++) {
        float a = A[i*width + k];
        float b = B[k*width + j];
        scalar += a * b;
    }
    C[i*width + j] = scalar;
}
```

# CUDA Keywords

Keyword	Ausführung	Aufruf
<u><i>__global__</i></u> void KernelFct()	Device	Host
<u><i>__device__</i></u> float DevFct()	Device	Device
<u><i>__host__</i></u> float HostFct()	Host	Host

# Umsetzung in CUDA C

---

- Initialisierung
  - Speicherallokation auf Host-Seite, Einlesen des Inputs
  - Speicherallokation auf Device-Seite: `cudaMalloc`
  - Kopieren des Inputs vom Host zum Device: `cudaMemcpy`
- GPU-Kernel starten
  - `MatrixMulKernel<<<grid, block>>>( . . . )`
- Abschluss
  - Kopieren des Outputs vom Device zum Host: `cudaMemcpy`
  - Speicher freigeben: `cudaFree`

# Abschluss

---

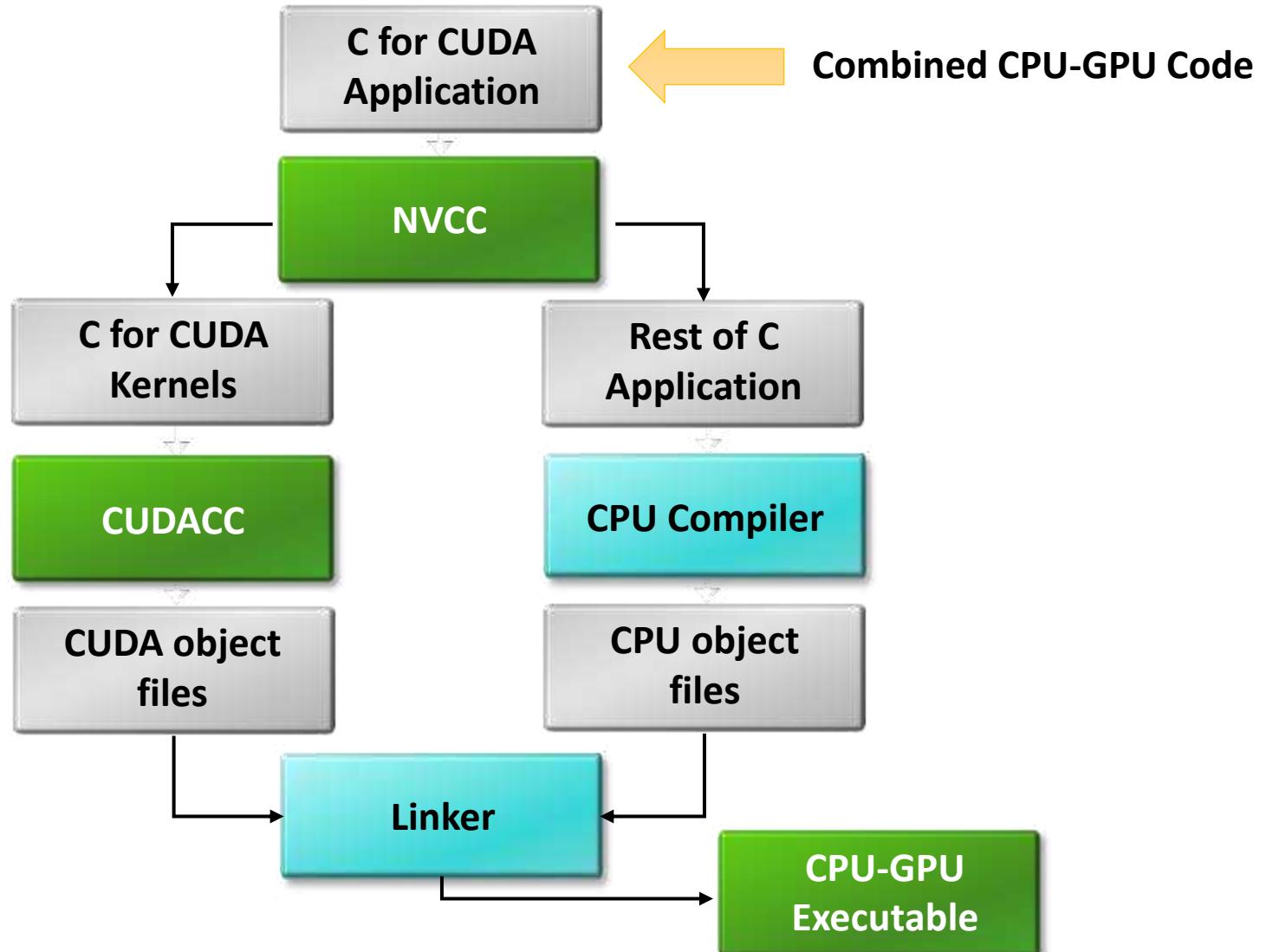
```
// Kopiere Ergebnismatrix vom Device zum Host  
cudaMemcpy(hostP, deviceP, size, cudaMemcpyDeviceToHost);  
  
// Gib Speicher auf Device frei  
cudaFree(deviceM);  
cudaFree(deviceN);  
cudaFree(deviceP);
```

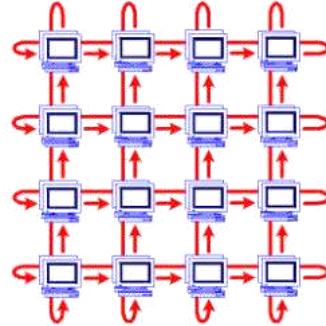
# Beispiel ist vereinfacht

---

- Voraussetzung an Matrizen
  - Quadratisch ( $n \times n$ )
  - Größe muss in einen Block passen und Vielfaches von 32 sein
  - $n \leq 16$
- Nur 1 Thread-Block
  - Beschränkt Größe auf  $16 \times 16$
- Erweiterung auf allgemeine Matrizen?

# CUDA-Programme





---

# Parallele Programmierung und Algorithmen

---

Tobias Lauer

Hochschule Offenburg

# Themen

---

- Paralleles Sortieren
  - *Teil I: Quicksort*
  - *Teil II: Radix Sort*
  - *Teil III: Mergesort (final)*
  - *Teil IV: Bitonic Sort (später)*

# Parallel Sort

---

- Sortieren von Daten
  - *wichtiger Baustein, allgegenwärtig*
    - Datenbanken
    - Suchmaschinen (Ranking)
    - ...
  - *Performance-Bottleneck*
    - $\Omega(n \log n)$  für allgemeine sequentielle Sortierverfahren
    - Zeit wächst stärker als linear mit der Datenmenge

→ Guter Kandidat für Parallelisierung

# Paralleles Quicksort

- Quicksort sequenziell bei Inputlänge  $n$ :
  - *In-place (kein Extraplatz nötig)*
  - *Komplexität*
    - $O(n \log n)$  im best und average case
    - $O(n^2)$  im worst case (ignorieren wir hier)
  - *Algorithmus:* *Zeit  $T(n)$ :*
    - Wähle ein Pivot-Element  $O(1)$
    - Teile das Array in:
      - **A**: Alle Elemente **kleiner** als Pivot
      - **B**: Pivot
      - **C**: Alle Elemente **größer/gleich** dem Pivot $O(n)$
    - Sortiere **A** und **C** rekursiv  $2 \cdot T(n/2)$

# Parallelisierung

- Algorithmus:
  - Wähle ein Pivot-Element
  - Teile das Array in:
    - A: Alle Elemente **kleiner** als Pivot
    - B: Pivot
    - C: Alle Elemente **größer/gleich** dem Pivot
  - Sortiere A und C rekursiv
- Naheliegend:
  - *Führe die beiden Sortierungen von A und C parallel aus*

# Laufzeit sequentiell

$$\begin{aligned}T(n) &= n + 2 \cdot T(n/2) \\&= n + 2 \underbrace{(n)}_{\sim} + 2 \cdot T(n/4) \\&= 2n + 4 \cdot T(n/4) \\&= 2n + 4 \cdot \left(\frac{n}{4} + 2 \cdot T(n/8)\right) \\&= 3n + 8 \cdot T(n/8) \\&\quad \dots \\&= \underbrace{n \cdot \log n}_{\sim} + \underbrace{n \cdot T(1)}_{\sim} \\&= O(n \cdot \log n)\end{aligned}$$

# Parallelisierung

- Algorithmus:
  - Wähle ein Pivot-Element
  - Teile das Array in:
    - A: Alle Elemente **kleiner** als Pivot
    - B: Pivot
    - C: Alle Elemente **größer/gleich** dem Pivot
  - Sortiere A und C rekursiv parallel
- Komplexität:
  - Zeitbedarf (Span):  $T_\infty(n) = O(n) + \textcolor{violet}{1} \cdot T(n/2) = ?$

Zeit parallel ① :

$$\begin{aligned} T(n) &= n + \overline{T}(n/2) \\ &= n + (n/2 + T(n/4)) \\ &= n + n/2 + n/4 + \overline{T}(n/8) \\ &= n + n/2 + n/4 + n/8 + T(n/16) \\ &= \dots \\ &= n + n/2 + n/4 + \dots + 1 + \overline{T}(1) \\ &\approx n \cdot \underbrace{\sum_{k=0}^{\log n} 1/2^k}_{= O(n)} \leq n \cdot \sum_{k=0}^{\infty} 1/2^k = 2n \end{aligned}$$

# Parallelisierung

- Algorithmus:
  - Wähle ein Pivot-Element
  - Teile das Array in:
    - A: Alle Elemente **kleiner** als Pivot
    - B: Pivot
    - C: Alle Elemente **größer/gleich** dem Pivot
  - Sortiere A und C rekursiv parallel
- Komplexität:
  - Zeitbedarf (Span):  $T_\infty(n) = O(n) + \textcolor{violet}{1} \cdot T(n/2) = O(n)$
  - Speedup ist im Bereich  $O(\log n)$
  - Wieviele Prozessoren sind nötig?

# Paralleles Quicksort

---

- D.h. eine Milliarde (=  $10^9$ ) Elemente würden ca. 30x schneller sortiert...
  - ... mit  $O(n)$  (z.B. 1/2 Milliarde) Prozessoren ... 😞
  - Können wir noch besser parallelisieren,  
z.B. auch den aufwendigeren Partitionsschritt?
- Ja, wenn wir extra Platz zulassen!

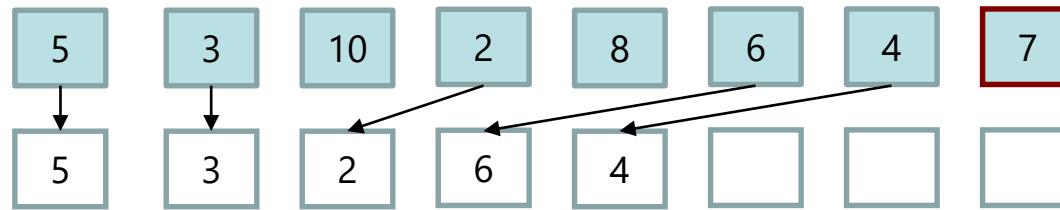
# Partition

---

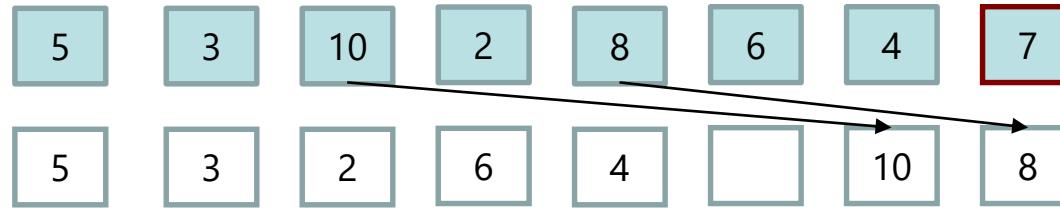
- Erinnerung
  - $A$ : Elemente kleiner als Pivot
  - $B$ : Pivot
  - $C$ : Elemente größer/gleich Pivot
- Sequenzielle Zeit:  $O(n)$
- Kann man  $A$  und  $C$  schneller aus dem Array filtern?
- Ja: führe 2x *Pack* aus!

# Pack

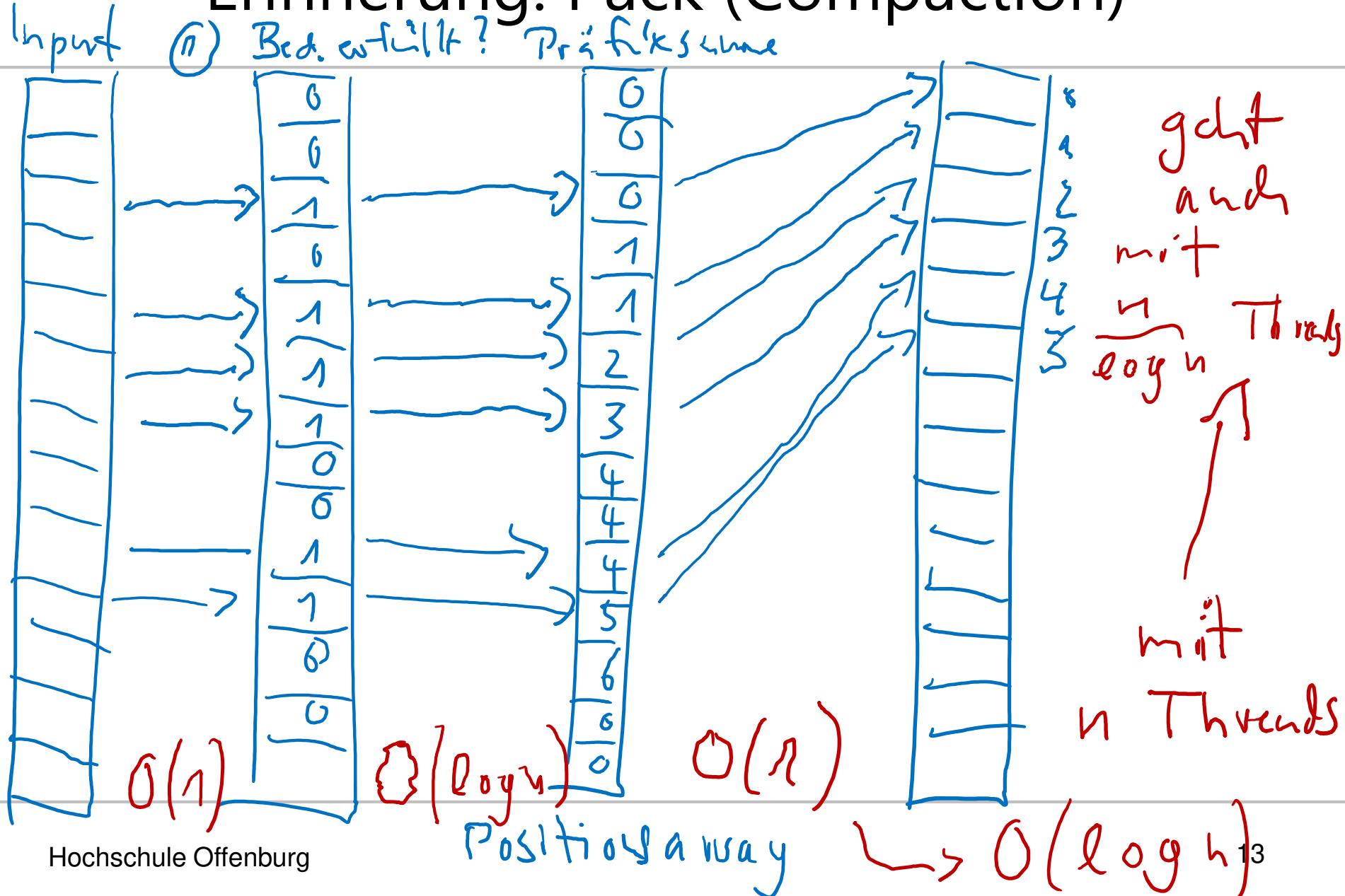
- Pack 1:
  - Bedingung:  $a[i] < pivot$



- Pack 2:
  - Bedingung:  $a[i] \geq pivot$



# Erinnerung: Pack (Compaction)



# Extra Speicherplatz

---

- Array zum Merken der Erfüllung der Bedingung
  - *Darauf Präfixsumme zur Berechnung der Zielposition*
- Hilfsarray für die Ergebnisse von Pack 1 und 2
  - können ihre Ergebnisse A und C ins selbe Hilfsarray schreiben (dazwischen noch Pivot einfügen!)
  - man kann Pack 1 und 2 auch zusammen durchführen (ändert nichts an asymptotischer Komplexität)
- Am Ende Hilfsarray mit Originalarray vertauschen (immer zwischen 2 Arrays hin- und her sortieren)

# Komplexität

---

- Partition
  - *2x Pack*
  - *Zeit (Span):  $O(\log n)$*
  - *Prozessoren:  $O(n / \log n)$*
- Gesamt:
  - *Zeit (Span):  $T(n) = O(\log n) + 1 \cdot T(n/2)$  =*
  - *Prozessoren:  $O( n / \log n )$*

# Parallel Quicksort: Fazit

---

- Quicksort kann *cost-optimal* parallelisiert werden
- Es gibt mehrere weitere Parallelisierungsvarianten

# Radix Sort

---

- Schnellstes bekanntes paralleles Sortierverfahren in der Praxis (auf GPUs)
  - vor allem für *große Anzahl* von Elementen ...
  - ... und *nicht zu großen* Elementen
- Beruht **nicht nur** auf Vergleichen und Vertauschen
  - Kein *allgemeines* Sortierverfahren
  - macht sich das verwendete Zahlenformat zunutze!

# Idee

---

- Sortiere durch iteriertes Verteilen auf „Fächer“, wobei es für jede Ziffer der Darstellung der Zahlen ein Fach gibt.
  - *Beginne bei der letzten Ziffer (ganz rechts)*
  - *Lasse die Reihenfolge der Elemente pro „Fach“ gleich*
- Wurde früher angewendet bei manuellem bzw. mechanischem Sortieren (z.B. Postleitzahlen, Lochkarten)

# Beispiel (Tafel)

---

# Durchführung

---

1. Wähle „passende“ Basis  $d$
2. Starte mit niederwertigster Stelle  $i$
3. Partitioniere Elemente je nach Wert an Stelle  $i$ 
  - *Dabei bisherige Reihenfolge zwischen den Elementen innerhalb der Gruppe nicht verändern*
4. Hänge die Partitionen aneinander  
→ neue globale Anordnung
5. Erhöhe  $i$  um 1
6. Falls noch nicht sortiert, gehe zu Schritt 3.

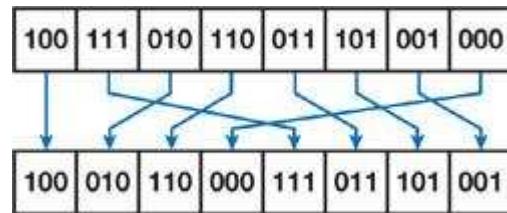
# Komplexität

---

- Anzahl Durchläufe:
  - bis zu  $e$  ( $e$  ist Exponent zur Darstellungsbasis)
  - $e$  ist für konkrete Anwendung konstant (aber relevant)
  - kann in bestimmtem Rahmen gewählt werden
  - bestimmt potentiell zusätzlichen Platzbedarf
- Anzahl Schritte pro Durchlauf:
  - $n$  Elemente anschauen und verteilen
- $O(e \cdot n)$

# Parallelisierung

- Zunächst:  $d = 2$  (Binärdarstellung)
- Beispiel: 4, 7, 2, 6, 3, 5, 1, 0 (3 Bits benötigt)
  - *Betrachte ersten Durchlauf:*



- *Wir splitten die Folge in zwei Teile:*
  - Elemente, die mit 0 enden
  - Elemente, die mit 1 enden
- *Wie können wir das tun?*

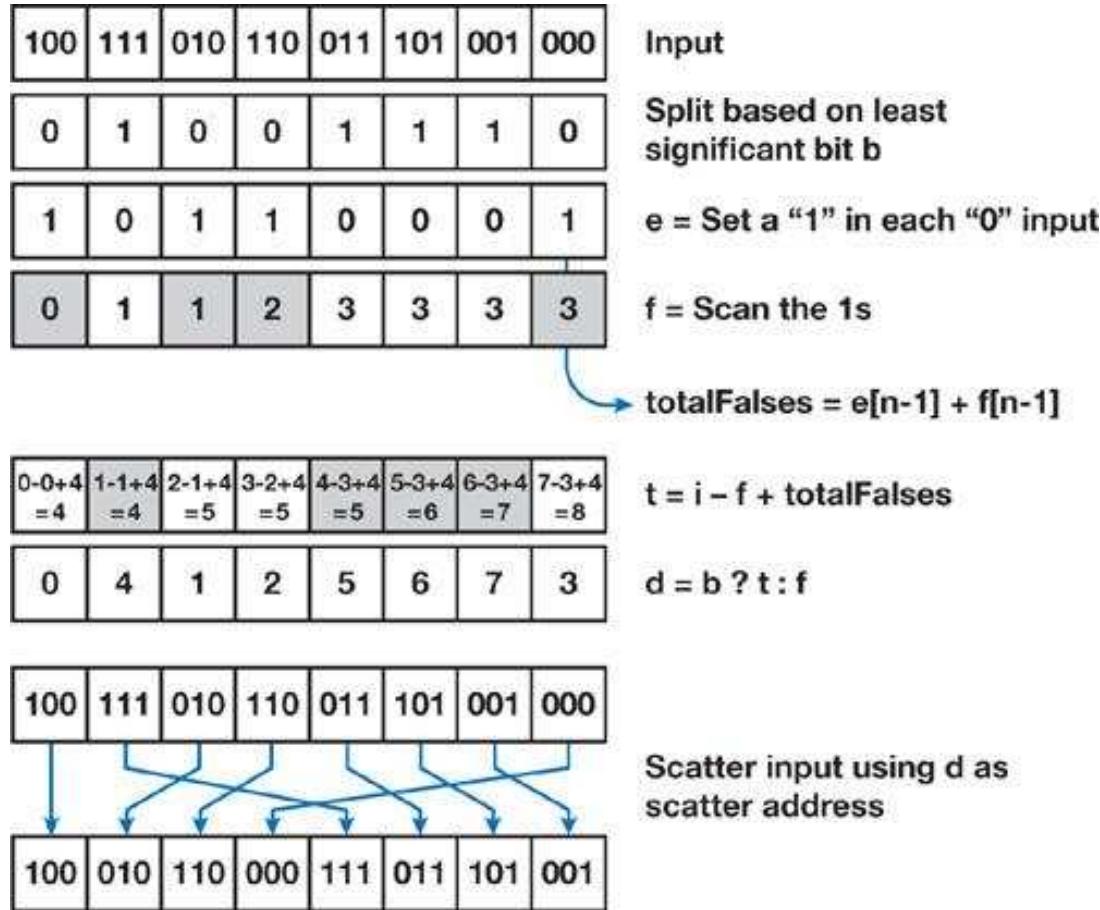
# Parallel Primitive: *Split*

---

- Partitioniere eine Folge anhand einer Bedingung in zwei Teilfolgen
- Kennen wir schon!
- Entspricht im Wesentlichen 2x *Pack*

# Split

- Wähle das relevante Bit im Input
- Speichere Inverses des betrachteten Bits in temporärem Array e
- Präfixsumme auf e → f: f enthält die Zieladressen der Elemente mit 0
- Speichere Gesamtzahl der 0-Werte in globaler Variable
- Weiteres temporäres Array t für Zielposition der 1-Werte
- Wähle d aus t bzw. f, je nach Bitwert
- Kopiere Inputelemente an Position d in Output



# Radix Sort mit Split

---

- Zur kompletten Sortierung:
  - *Wiederhole Split mit dem nächsthöheren Bit ...*
  - *... solange, bis die Folge sortiert ist.*
  - *Wann ist das?*
    - Spätestens nach dem Split mit dem höchstwertigen Bit
    - Wenn die Schlüssel nicht zu groß sind, ggfs. auch schon viel früher
- Komplexität:
  - *Zeit pro Split:*       $O(\log n)$

# Radix Sort mit Split

---

- Zur kompletten Sortierung:
  - *Wiederhole Split mit dem nächsthöheren Bit ...*
  - *... solange, bis die Folge sortiert ist.*
  - *Wann ist das?*
    - Spätestens nach dem Split mit dem höchstwertigen Bit
    - Wenn die Schlüssel nicht zu groß sind, ggfs. auch schon früher
- Dennoch Problem: viele Durchgänge!
  - *Ausweg: s. Labor*

# Praktische Aspekte

---

- Implementierung für sehr großen Input?
  - *Aufteilung in Blöcke der Größe eines SIMD-Arrays*
  - *Sortierung pro Block*
  - *Zusammenführen der sortierten Blöcke*  
→ Parallel merge

# Merge-Sort in der Praxis

---

- Problem:  
Baumartige Aufteilung in immer kleinere Teilfolgen
- Führt zu verschiedenen Szenarien:
  - *Mehr Merges als Threads im unteren Teil des Baums*
  - *Merges ~ Threads in einer mittleren Region*
  - *Mehr Threads als Merges im oberen Teil*
- Idee:  
Nutze verschiedene Ansätze je nach Region im Baum

# Merge-Methoden

---

- Im unteren Teil:
  - *Sequenziell:*  
*Jeder Thread verschmilzt kleine Teilfolgen sequenziell  
(bzw. sortiert diese anderweitig sequenziell)*
  - *Mittlerer Teil:*  
*Parallelisierung gemäß Architektur (z.B. GPU-Multiprozessoren)*
    - 1 oder wenige Merges pro Thread
    - ermöglicht optimale parallele Berechnung unabhängiger Teilfolgen  
(z.B. pro Prozessorgruppe)
  - *Oberer Teil:*
    - Einzelnen Merge-Schritt parallelisieren  
(→ siehe später: Parallel Merge)

# Sortier-Netzwerke

---

- Eigenschaft von vielen Sortieralgorithmen
  - *Datenabhängigkeit*
  - *Nächste Aktion hängt vom Wert des betrachteten Schlüssels ab*
  - *Threads sind ungleichmäßig ausgelastet / divergieren*
  - *Nicht optimal für SIMD-artige Architekturen*
- Besser geeignete Algorithmen
  - *Möglichst gleiche Aufgabe für alle Threads*  
→ „Sorting Networks“

# Bitonic Sort

---

- Ähnlichkeiten zu Merge-Sort
  - *Rekursiver Ansatz*
    - Halbierung der Folge in jedem Schritt
      - Sortieren der Hälften wieder mit Bitonic Sort
    - **Neu:** Verschmelzen der sortierten Teilfolgen ist ebenfalls rekursiv
      - „Bitonic Merge“
  - *aber: in-place (kein extra Platz benötigt)*
    - kein Kopieren, nur Vertauschen
- Nutzt eine wichtige Eigenschaft von **bitonischen Folgen**

# Bitonische Zahlenfolgen

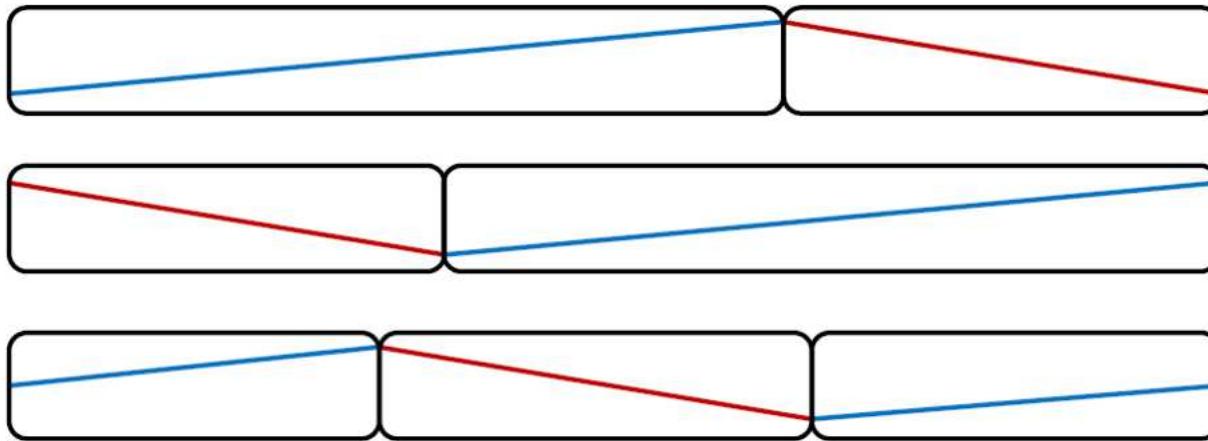
---

- Bitonic (vgl. monotonic):
  - Folge besteht aus einem *monoton steigenden* und einem *monoton fallenden* Teil
  - Bsp.

7, 10, 24, 21, 18, 10, 4, 3

# Definition

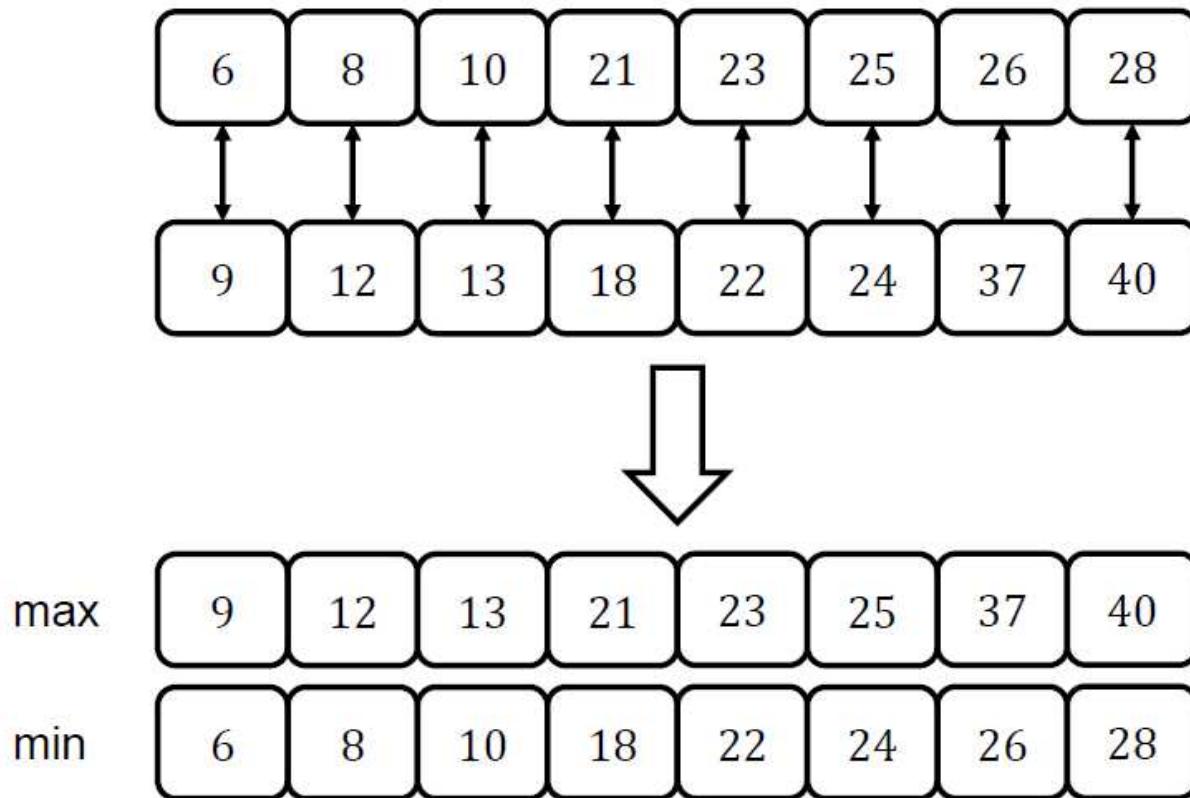
- Eine Folge  $e_0, e_1, \dots, e_n$  heißt **bitonisch**, wenn es einen Index  $i$  ( $0 < i < n$ ) gibt, so dass
  - entweder  $e_0, \dots, e_i$  monoton steigend und  $e_i, \dots, e_n$  monoton fallend ist
  - oder es eine zyklische Verschiebung der Folge gibt, für die obiges gilt.
- Bsp:



Bildquelle: Philipp Slusallek

# Comparison Network

- Elementweiser Vergleich von zwei Folgen (hier: sortiert)



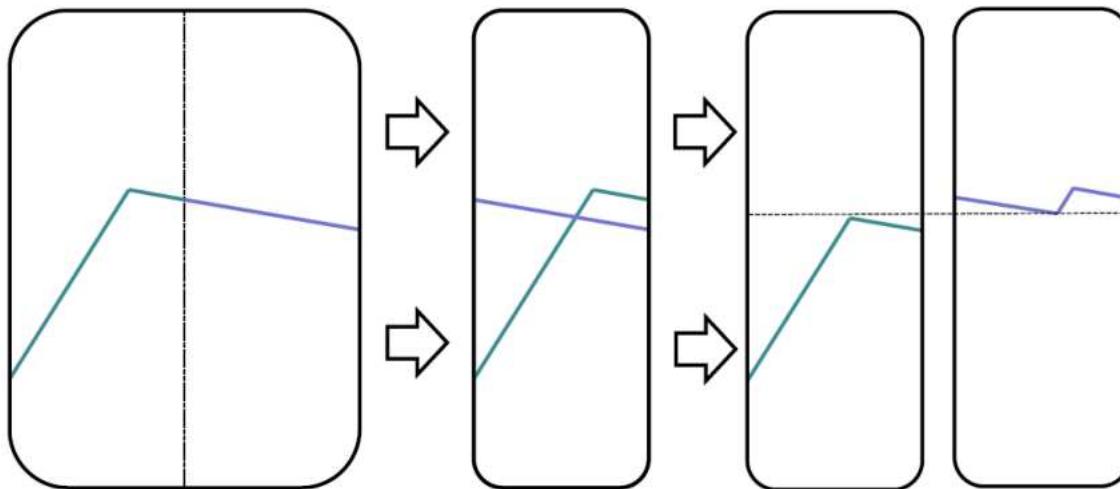
# Comparison Network

---

- Elementvergleiche
    - *Komplett unabhängig voneinander*
    - *Immer gleich viele (nämlich  $n/2$ )*
    - *Immer gleich strukturiert (unabhängig von Datenbeschaffenheit)*  
→ *Perfekt parallelisierbar („embarrassingly parallel“)*
- Sehr gut geeignet für SIMD, z.B. für GPUs

# Beobachtung 1

- Zerlegung einer **bitonischen** Folge in 2 Teilfolgen (gleicher Länge):

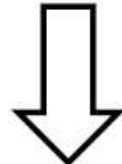


- Comparison-Schritt: beide Teilfolgen sind danach ebenfalls wieder **bitonisch** (Beweis nicht trivial!)
- Alle Elemente der **rechten Teilfolge** sind größer als alle Elemente der **linken Teilfolge**

# Bitonic Split

- Teilt eine Folge durch ein Comparison Network:

BitonicSplit(bs[2\*N])



$S1[N] = \{ \min( bs[0], bs[N] ), \dots, \min( bs[N-1], bs[2*N-1] ) \}$

$S2[N] = \{ \max( bs[0], bs[N] ), \dots, \max( bs[N-1], bs[2*N-1] ) \}$

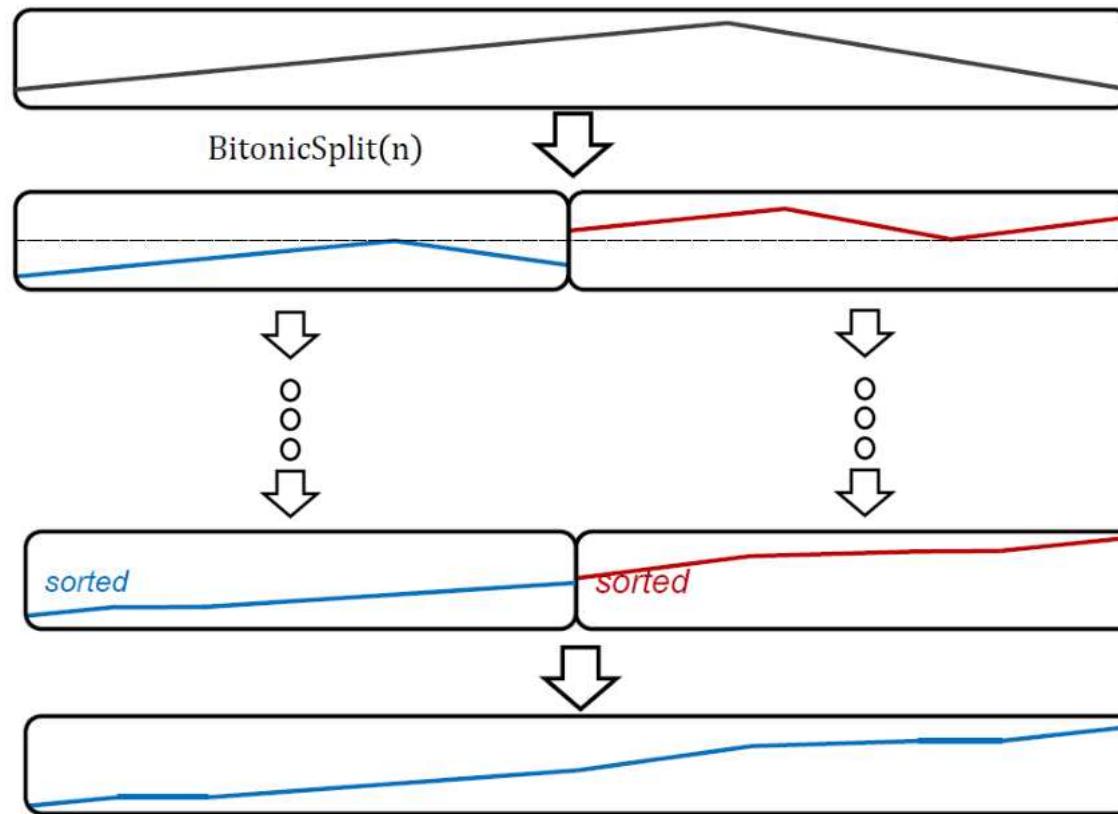
# BitonicSplit

---

```
bitonicSplit(int lo, int hi) {  
    if (lo >= hi) return;  
    int m = (hi+lo)/2;  
    int stride = (hi-lo+1)/2; // Abstand verglichener Elemente  
    for (int i=lo; i<=m; i++) in parallel {  
        if (input[i] > input[i+stride]) {  
            swap(i, i+stride);  
        }  
    }  
}
```

# Beobachtung 2

- Durch wiederholtes (rekursives) Splitten wird eine **bitonische** Folge komplett **sortiert** ( $\rightarrow$  „BitonicMerge“)



# BitonicMerge (rekursiv)

---

```
bitonicMerge(int lo, int hi) {  
    if (lo >= hi) return;  
    bitonicSplit(lo, hi);  
    int m = (hi+lo)/2;  
    bitonicMerge(lo, m);  
    bitonicMerge(m+1, hi);  
}
```

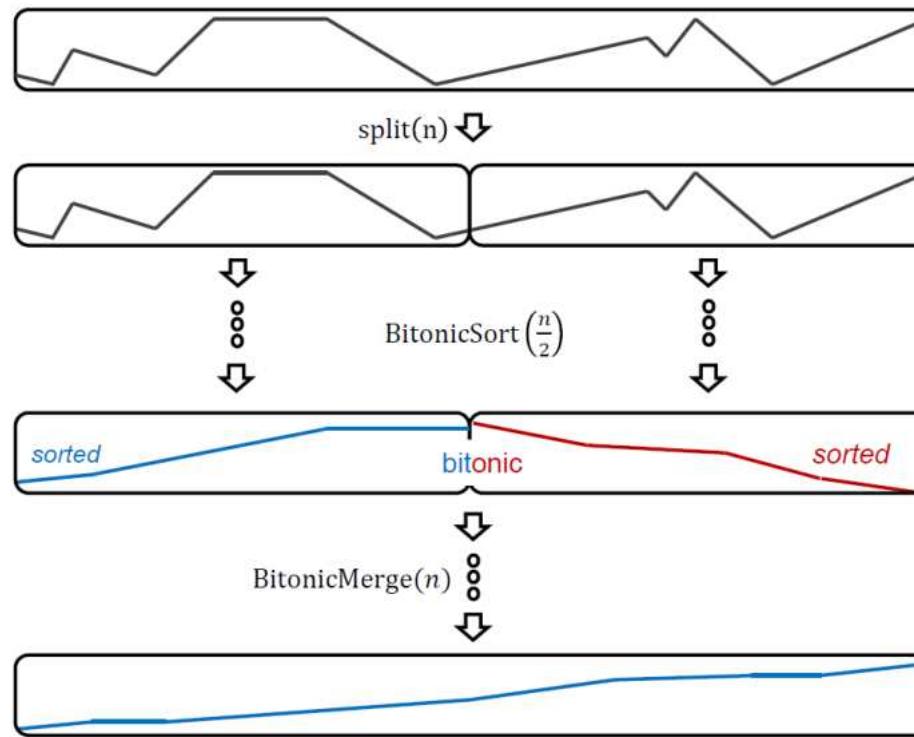
# Was tun bei nicht-bitonischen Folgen?

---

- Üblicherweise sind **unsortierte** Folgen **nicht bitonisch**
- Kann man sie bitonisch machen?
  - *Idee:*
    - Teile die unsortierte Folge in zwei gleich große Hälften
    - Sortiere die **linke Hälfte aufsteigend** und die **rechte absteigend** (wie? → rekursiv mit BitonicSort)
    - Die Folge ist nun **bitonisch** und kann nun mit einem **BitonicMerge** sortiert werden.

# BitonicSort

- BitonicSort( $n$ )

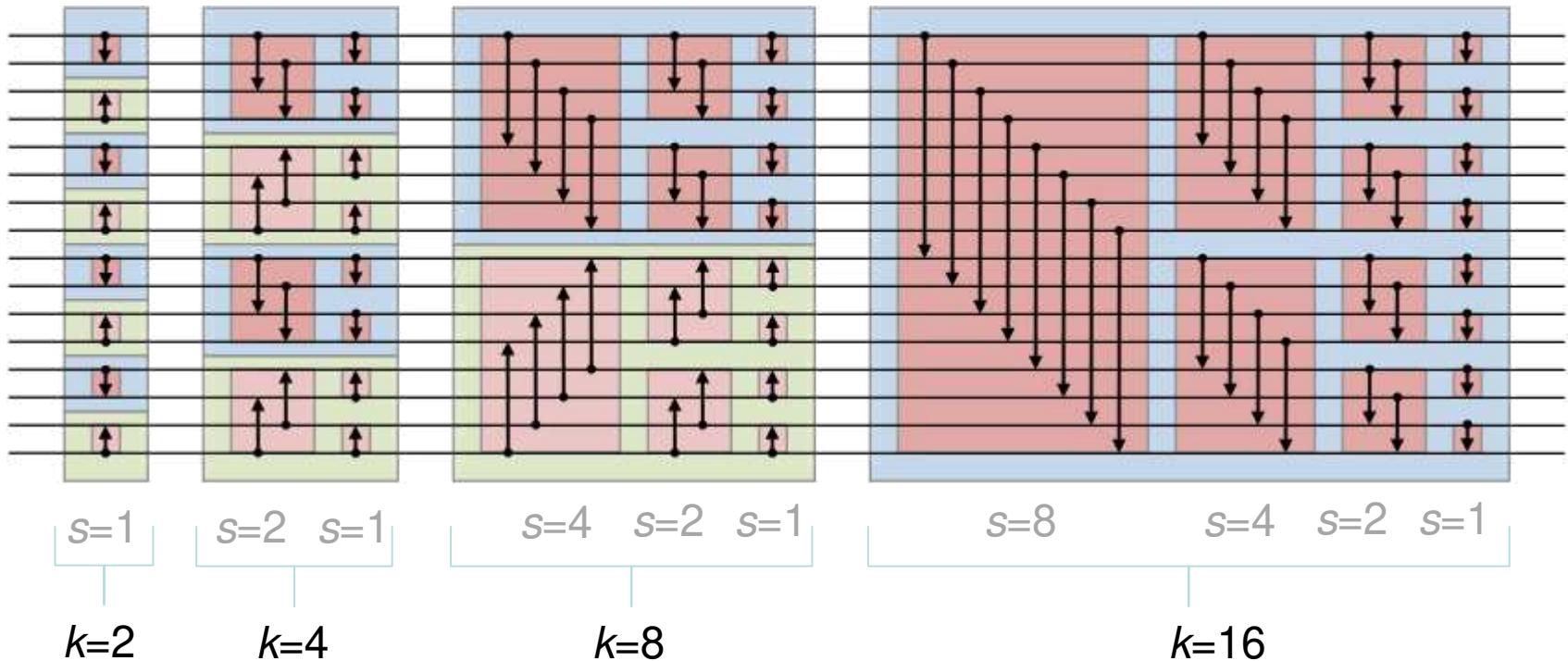


# BitonicSort (rekursiv)

```
static void bitonicSort(int lo, int hi) {  
    if (lo >= hi) return;  
    // Split the input  
    int m = (hi+lo)/2;  
    // Sort lower half in same order  
    bitonicSort(Lo, m);  
    // Sort upper half in reverse order  
    bitonicSort_reverse(m+1, hi);  
    // Sort bitonic sequence lo..hi  
    bitonicMerge(Lo, hi);  
}
```

# Für GPU: Rekursion auflösen

- Iterative Sichtweise (Bsp. mit 16 Elementen):



$k$ : aktuelle Größe der Teilfolge(n)     $s$ : Abstand zwischen zwei verglichenen Elementen

# Iterativer Algorithmus

```
bitonicSort_iterative(int[] input) {  
    final int NUM = input.length;  
    for (int k=2; k<=NUM; k*=2) {  
        for (int s=k/2; s>0; s/=2) {  
            for (int tid=0; tid<NUM; tid++) in parallel {  
                int ixj = tid ^ s; // XOR  
                if (ixj > tid) {  
                    if ((tid & k) == 0) {  
                        if (input[tid] > input[ixj])  
                            swap(input, tid, ixj);  
                    } else {  
                        if (input[tid] < input[ixj])  
                            swap(input, tid, ixj);  
                    }  
                }  
            } // sync  
        }  
    }  
}
```

// log<sub>2</sub>(n) Iterationen  
// <= log<sub>2</sub>(n) Iterationen

// Tauschpartner bestimmen  
// nur 1 Tausch pro 2 Elemente  
// In „gerader k-Gruppe“ ...  
// ... aufsteigend sortieren

// sonst...  
// ... absteigend sortieren

# Komplexität

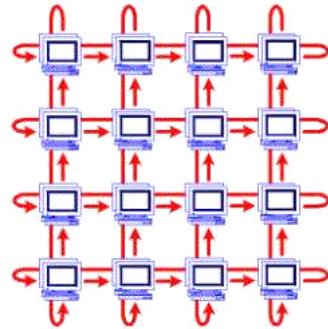
---

- Cost:  $O(n \log^2 n)$
- Time/Span:  $O(\log^2 n)$
- Prozessoren:  $O(n)$

# Zusammenfassung

---

- Parallelisierung kann Sortierung stark beschleunigen
- Allgemeine Sortierverfahren (Quicksort, Merge-Sort) sind flexibel in Bezug auf die zu sortierenden Elemente
- Spezielle Verfahren wie Radix-Sort können sehr schnell sein (~1 Mrd. 32-bit-Zahlen pro Sekunde auf einer GPU).
- Bitonic Sort hat keine Datenabhängigkeit und daher minimale Thread-Divergenz



---

# Parallel Programming and Algorithms

Tobias Lauer

Hochschule Offenburg

---

# Topics

---

- Parallele Algorithmen
  - *Parallel Merge*  
*(Vereinigung von 2 großen sortierten Arrays)*

# Parallel Merge

---

- *Merging* kennen wir schon von Merge-Sort
- Input:  
Zwei sortierte Arrays  $A$  und  $B$  der Längen  $m$  und  $n$ 
  - Hier: *nicht unbedingt gleiche/ähnliche Länge*
  - Wir nehmen an, dass  $m \geq n$  (ansonsten vertausche  $A$  und  $B$ )
- Output:  
Ein Array  $C$  der Länge  $m+n$ , das die Elemente von  $A$  und  $B$  sortiert enthält

# Beispiel

$$A = (2, 8, 11, 13, 17, 20) \quad B = (3, 6, 10, 15, 16, 73)$$

$$\rightarrow C = (2, 3, 6, 8, 10, 11, 13, 15, 16, 17, 20, 73)$$

- Sequenzieller Algorithmus (siehe Merge-Sort)
  - Gehe  $A$  und  $B$  simultan mit Hilfe zweier Zeiger durch
  - In jedem Schritt:
    - Kopiere das jeweils kleiner der beiden Elemente aus  $A$  bzw.  $B$  an die aktuelle Position in  $C$
    - Erhöhe den entsprechenden Zeiger und die aktuelle Position in  $C$
  - Die Komplexität ist  $O(m+n)$

# Parallelisierung: 1. Versuch

---

- Beobachtung:
  - *Man kann das Merging von links (aufsteigend) oder von rechts (absteigend) durchführen*
  - *Geht auch beides gleichzeitig?*
    - Ja, wenn man das Zielarray jeweils genau bis zur Hälfte füllt (dann kann es keine Schreibkonflikte geben)
  - *Paralleles Merging mit 2 Threads/Prozessoren möglich*
    - Mehr geht mit diesem Ansatz nicht!
- Geht noch „mehr“ Parallelität?

# Paralleles Merging

---

Idee:

Teile die Arrays geeignet auf und führe den sequenziellen Algorithmus parallel auf den Teilen aus

Definition:

$\text{rank}(a_i : A)$  = Anzahl von Elementen in  $A$ ,  
die kleiner oder gleich  $a_i$  sind ( $a_i \in A$ )

$\text{rank}(b_i : B)$  = Anzahl von Elementen in  $B$ ,  
die kleiner oder gleich  $b_i$  sind ( $b_i \in B$ )

# Beispiel und Bemerkung

---

$$A = (2, 8, 11, 13, 17, 20) \quad B = (3, 6, 10, 15, 16, 73)$$

$$\text{rank}(11 : A) = 3$$

$$\text{rank}(11 : B) = 3$$

$$\text{rank}(16 : A) = 4$$

$$\text{rank}(16 : B) = 5$$

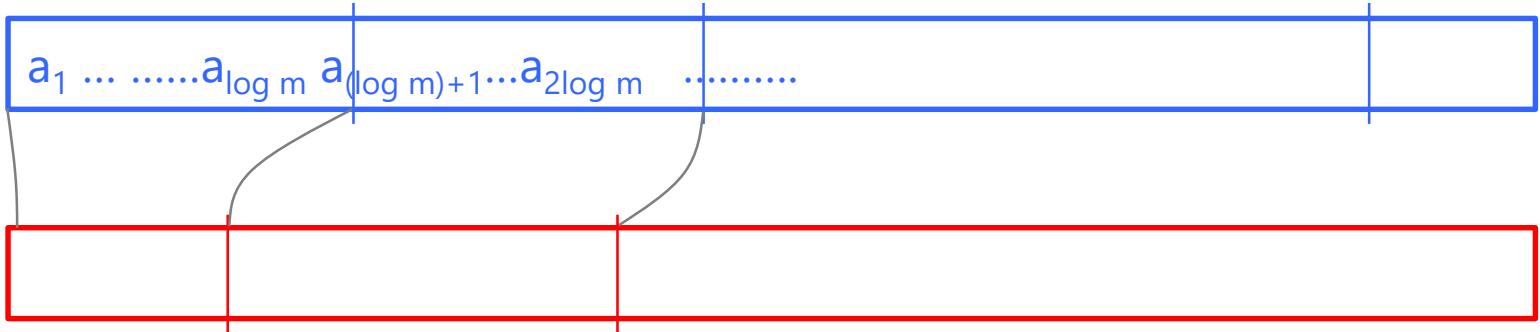
Bemerkung:

Der Rang eines Elements  $e$  aus  $A$  oder  $B$  im Ergebnisarray  $C$  entspricht

$$\text{rank}(e : A) + \text{rank}(e : B)$$

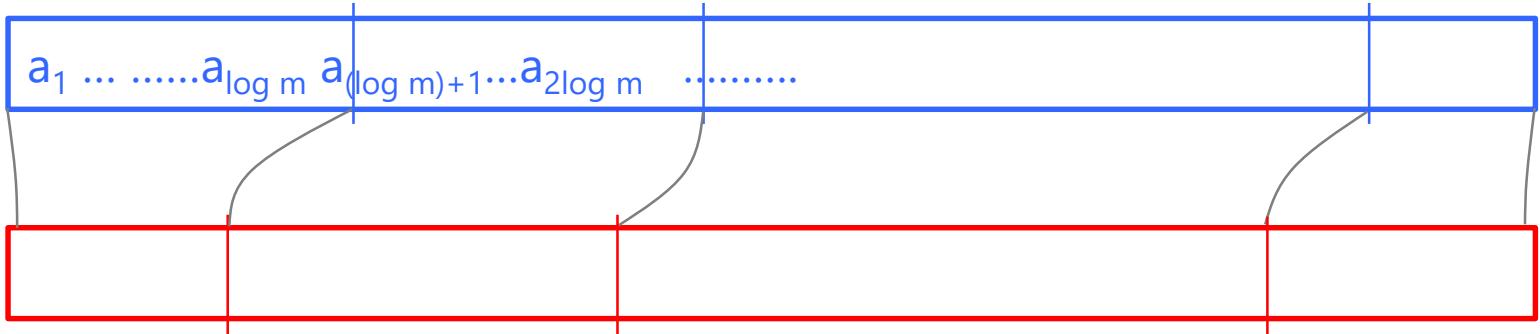
(wenn man bei 1 anfängt zu zählen)

# Aufteilung



- Teile Array  $A$  in Blöcke so dass jeder Block aus  $\log m$  Elementen besteht (außer evtl. dem letzten).
- Damit haben wir  $m/\log m$  Blöcke
- Der  $i$ -te Block endet an Position  $i \cdot \log m$  ( $1 \leq i < m/\log m$ )  
(und der letzte Block endet an Position  $m$ )

# Aufteilung



- Es gibt eine eindeutige Zuordnung jedes Blocks  $A_i$  zu einem Block in  $B$
- Wir nennen so ein Paar von Blöcken *Matching Pair*
- Problem: Wie findet man ein *Matching Pair*?

# Paralleles Merging

---

## Ziel:

- Laufzeit des sequenziellen Algorithmus für ein *Matching Pair* in  $O(\log m + \log n)$
- Dafür sollte die Größe jedes Teilstücks in  $O(\log m)$  bzw.  $O(\log n)$  sein
- Für *A* war diese Aufteilung leicht; was ist mit *B*?

## Ansatz:

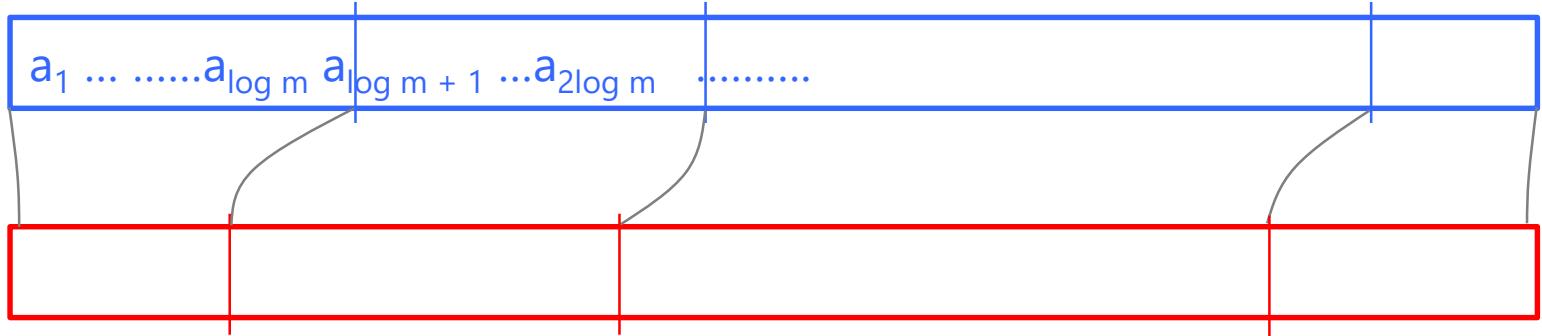
- Benutze den *Rang (rank)* von Elementen zur Aufteilung von *B* in passende Stücke
- Führe dann den sequenziellen Algorithmus *parallel* auf je einem *Matching Pair* aus

# Prozessoraufteilung

---

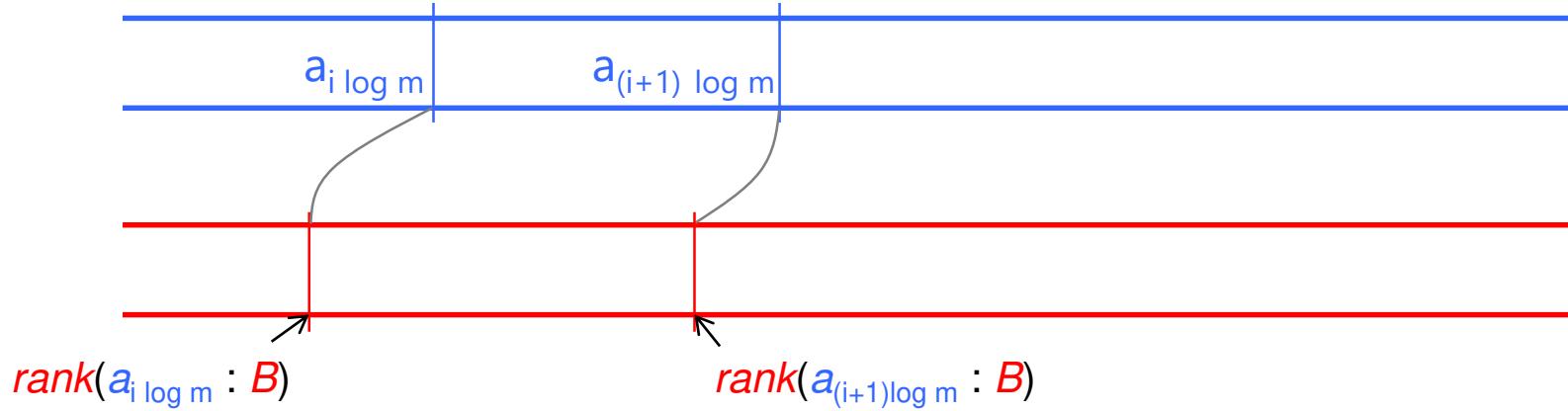
- Verwende für jeden Block  $A_i$  von  $A$  einen Prozessor
- Für alle  $i$  parallel (außer dem letzten):
  - Finde für das letzte Element von  $A_i$  seinen Rang in  $B$   
d.h. bestimme  $\text{rank}(a_{i \log m} : B)$
  - Wie kann man den Rang effizient bestimmen?
    - Binäre Suche
  - Zeitkomplexität für diesen Schritt
    - $O(\log n)$
  - Binäre Suchen können parallel stattfinden (CREW)
- Nach diesem Schritt ist auch Array  $B$  implizit in  $m/\log m$  Stücke unterteilt

# Aufteilung



- Es gibt eine eindeutige Zuordnung jedes Blocks  $A_i$  zu einem Block in  $B$  (Blöcke in  $B$  können auch leer sein)
- Wir nennen so ein Paar von Blöcken *Matching Pair*

# Aufteilung



- Der Rang jedes Elements innerhalb eines *Blocks  $A_i$*  liegt im korrespondierenden *Matching Block  $B_i$* .
- Daher ist das *Mergen* eines *Matching Pair* ein unabhängiges Teilproblem.

# Merging von Matching Pairs

---

Größe der Blöcke in *B*:

- Guter Fall:

- Größe jedes Blocks ist (ungefähr) in  $O(\log n)$   
→ Benutze den sequentiellen Algorithmus zum Merging

- Schlechter Fall:

- es gibt einen oder mehrere lange Blöcke  
→ Das müssen wir „reparieren“ (wie?)

# Worst case

---

- Ein einziger Block  $A_i$  von  $A$  hat als *Matching* Partner das komplette Array  $B$
- Idee: Verwende denselben Algorithmus für  $B$  und  $A_i$

# Guter Fall

---

- Verwende einen Prozessor pro Matching Pair
- Verschmilz die beiden Blöcke des Paars mit dem sequenziellen Algorithmus
  - *Die Zeit dafür ist in  $O(\log m + \log n)$*
- Alle Paare können gleichzeitig bearbeitet werden

# Komplexitätsanalyse

---

- Schritt 1: Aufteilung von Array  $A$ 
  - Die Aufteilung geht *rein nach Position (Index)*
  - Wir haben  $m/\log m$  Prozessoren  
(einen Prozessor pro Block der Größe  $\log m$ )
  - Daher lässt sich das Endelement jedes Blocks in konstanter Zeit  $O(1)$  bestimmen
  - Die Cost ist in  $O(m/\log m)$

# Komplexitätsanalyse

---

- Schritt 2: Aufteilung von Array  $B$ 
  - Diese Aufteilung geschieht nach Werten ( $\text{rank}(a : B)$ )
  - Wir benutzen dafür die binäre Suche in  $B$   
Diese benötigt Zeit in  $O(\log n)$
  - Auch hier haben wir  $m/\log m$  Prozessoren
  - Cost:  $O(\log n \cdot (m/\log m))$

# Abschätzung

---

$O(\log n \cdot (m/\log m))$

$$\begin{aligned}\frac{m \cdot \log n}{\log m} &\leq m \cdot \frac{\log(m+n)}{\log m} \\ &\leq m \cdot \frac{m+n}{m} = m+n\end{aligned}$$

Es gilt (außer für sehr kleine Werte von  $m$  und  $n$ ):

$$\frac{\log(m+n)}{\log m} < \frac{m+n}{m}$$

# Komplexitätsanalyse

---

- Schritt 2: Aufteilung von Array  $B$ 
  - Diese Aufteilung geschieht nach Werten ( $\text{rank}(a : B)$ )
  - Wir benutzen dafür die binäre Suche in  $B$   
Diese benötigt Zeit in  $O(\log n)$
  - Auch hier haben wir  $m/\log m$  Prozessoren
  - Cost:  $O( (m/\log m) \cdot \log n ) = O(m + n)$

# Komplexitätsanalyse

---

- Schritt 3: Sequenzielles Merging von Matching Pairs
  - Die Zeit dafür ist in  $O(\log m + \log n)$
  - Auch hier haben wir  $m/\log m$  Prozessoren
  - Cost:  
$$\begin{aligned} & O((m/\log m) \cdot (\log m + \log n)) \\ &= O(m + (m/\log m \cdot \log n)) \\ &= O(m) + O(m + n) \\ &= O(m + n) \end{aligned}$$

# Komplexitätsanalyse

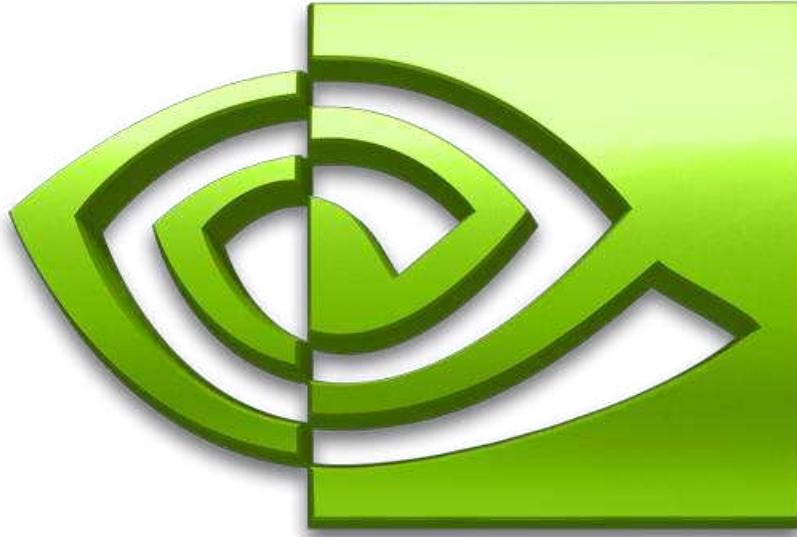
- Zusammenfassung:

	<i>Time</i>	<i>Cost</i>
<i>Schritt 1:</i>	$O(1)$	$O(m/\log m)$
<i>Schritt 2:</i>	$O(\log n)$	$O(m + n)$
<i>Schritt 3:</i>	$O(\log m + \log n)$	$O(m + n)$
<b>Gesamt:</b>	$O(\log m + \log n)$	$O(m + n)$

# Was ist mit den *bad cases*?

---

- Worst case:  
Ein einziger Block  $A_i$  von  $A$  hat als *Matching* Partner das komplette Array  $B$
- Dann verwende denselben Algorithmus für  $B$  und  $A_i$ :
  - Länge von  $A_i = \log m < m$
  - Länge von  $B = n$
  - Gesamlaufzeit für diesen Teil nie schlechter als vorher
- Für Fälle, die „dazwischen“ liegen, kann man analog vorgehen.



**nVIDIA.**<sup>®</sup>

# **Optimizing Parallel Reduction in CUDA**

**Mark Harris**  
**NVIDIA Developer Technology**

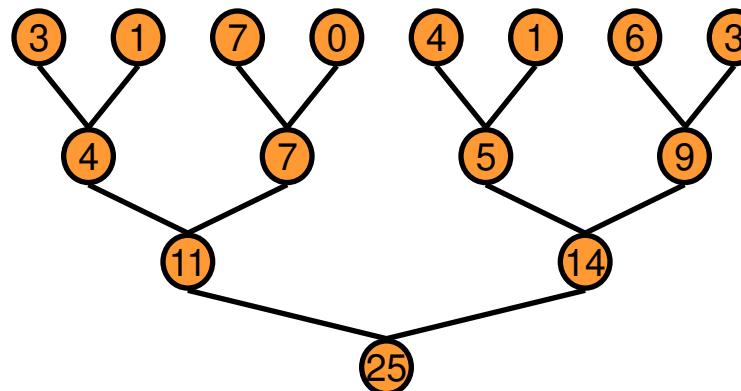
# Parallel Reduction



- ➊ Common and important data parallel primitive
- ➋ Easy to implement in CUDA
  - ➌ Harder to get it right
- ➌ Serves as a great optimization example
  - ➍ We'll walk step by step through 7 different versions
  - ➎ Demonstrates several important optimization strategies

# Parallel Reduction

- Tree-based approach used within each thread block



- Need to be able to use multiple thread blocks
  - To process very large arrays
  - To keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array
- But how do we communicate partial results between thread blocks?

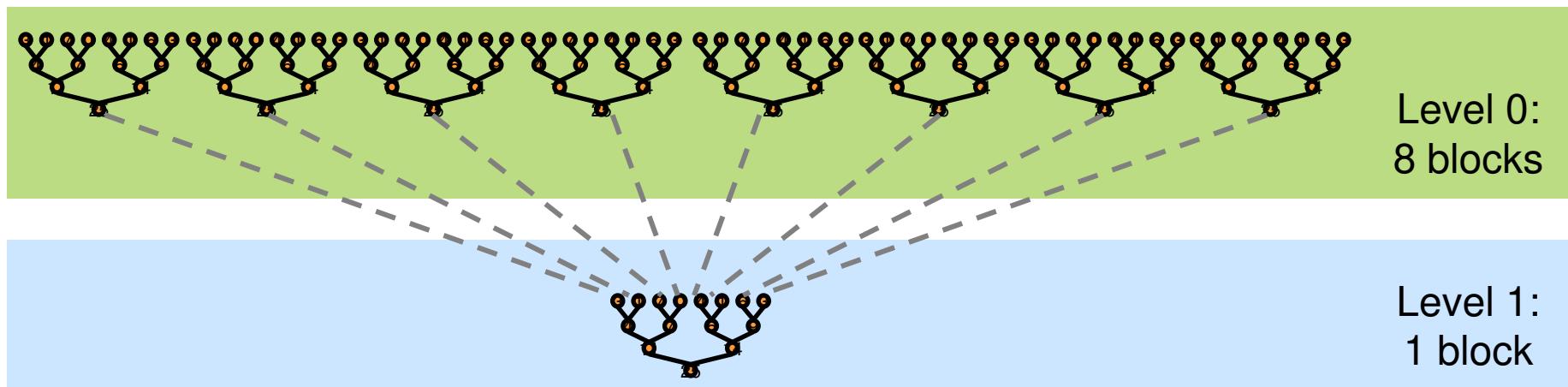


# Problem: Global Synchronization

- If we could synchronize across all thread blocks, could easily reduce very large arrays, right?
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively
- But CUDA has no global synchronization. Why?
  - Expensive to build in hardware for GPUs with high processor count
  - Would force programmer to run fewer blocks (no more than # multiprocessors \* # resident blocks / multiprocessor) to avoid deadlock, which may reduce overall efficiency
- Solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible HW overhead, low SW overhead

# Solution: Kernel Decomposition

- Avoid global sync by decomposing computation into multiple kernel invocations



- In the case of reductions, code for all levels is the same
  - Recursive kernel invocation



# What is Our Optimization Goal?

- ➊ We should strive to reach GPU peak performance
- ➋ Choose the right metric:
  - ➌ GFLOP/s: for compute-bound kernels
  - ➌ Bandwidth: for memory-bound kernels
- ➌ Reductions have very low arithmetic intensity
  - ➌ 1 flop per element loaded (bandwidth-optimal)
- ➍ Therefore we should strive for peak bandwidth
  
- ➎ Will use G80 GPU for this example
  - ➏ 384-bit memory interface, 900 MHz DDR
  - ➏  $384 * 1800 / 8 = 86.4 \text{ GB/s}$



# Reduction #1: Interleaved Addressing

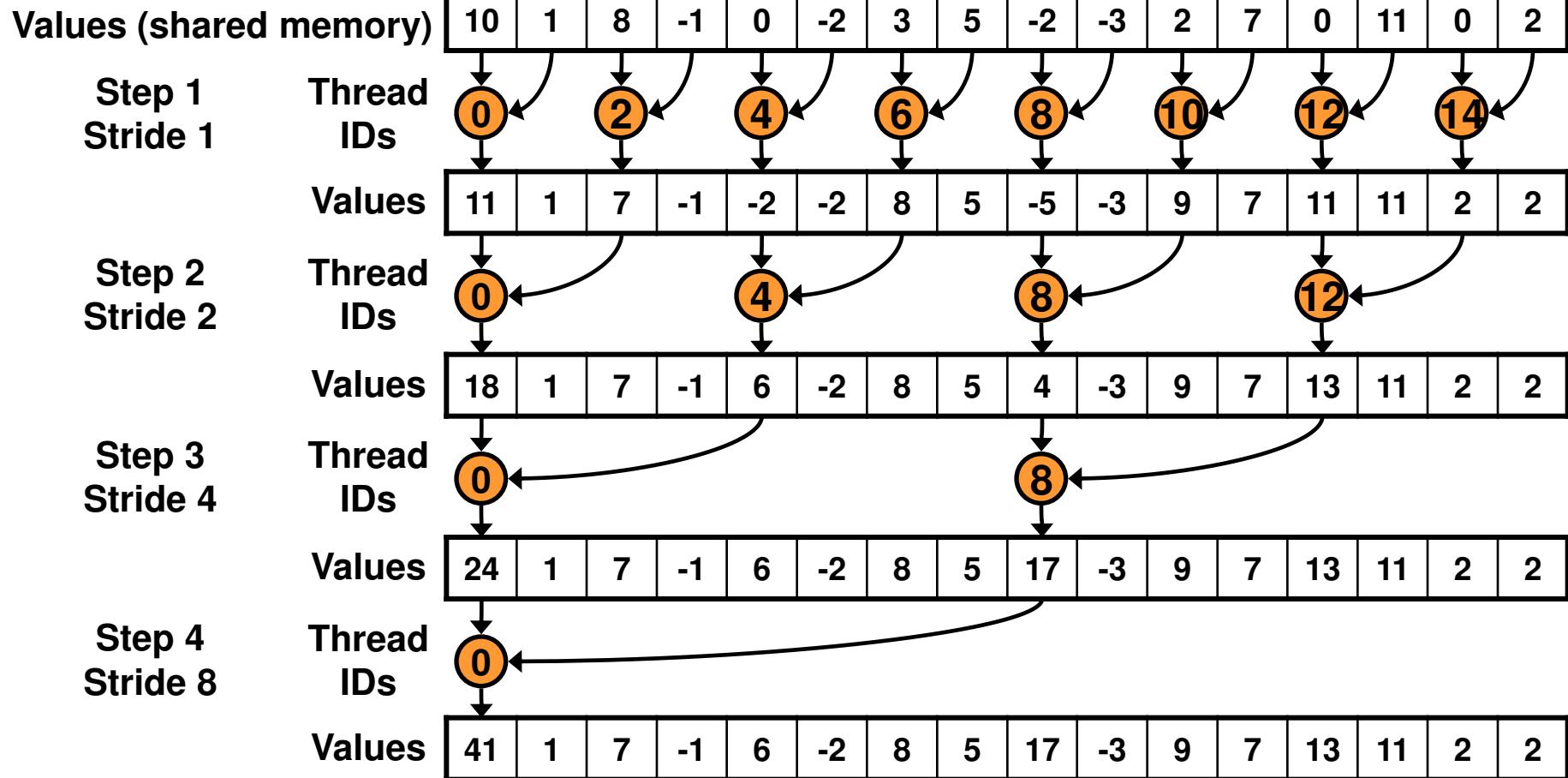
```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

# Parallel Reduction: Interleaved Addressing





# Reduction #1: Interleaved Addressing

```
__global__ void reduce1(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();
```

```
// do reduction in shared mem
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) { ←
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

**Problem:** highly divergent warps are very inefficient, and % operator is very slow

```
// write result for this block to global mem
if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

# Performance for 4M element reduction



	Time ( $2^{22}$ ints)	Bandwidth
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>

Note: Block Size = 128 threads for all tests



# Reduction #2: Interleaved Addressing

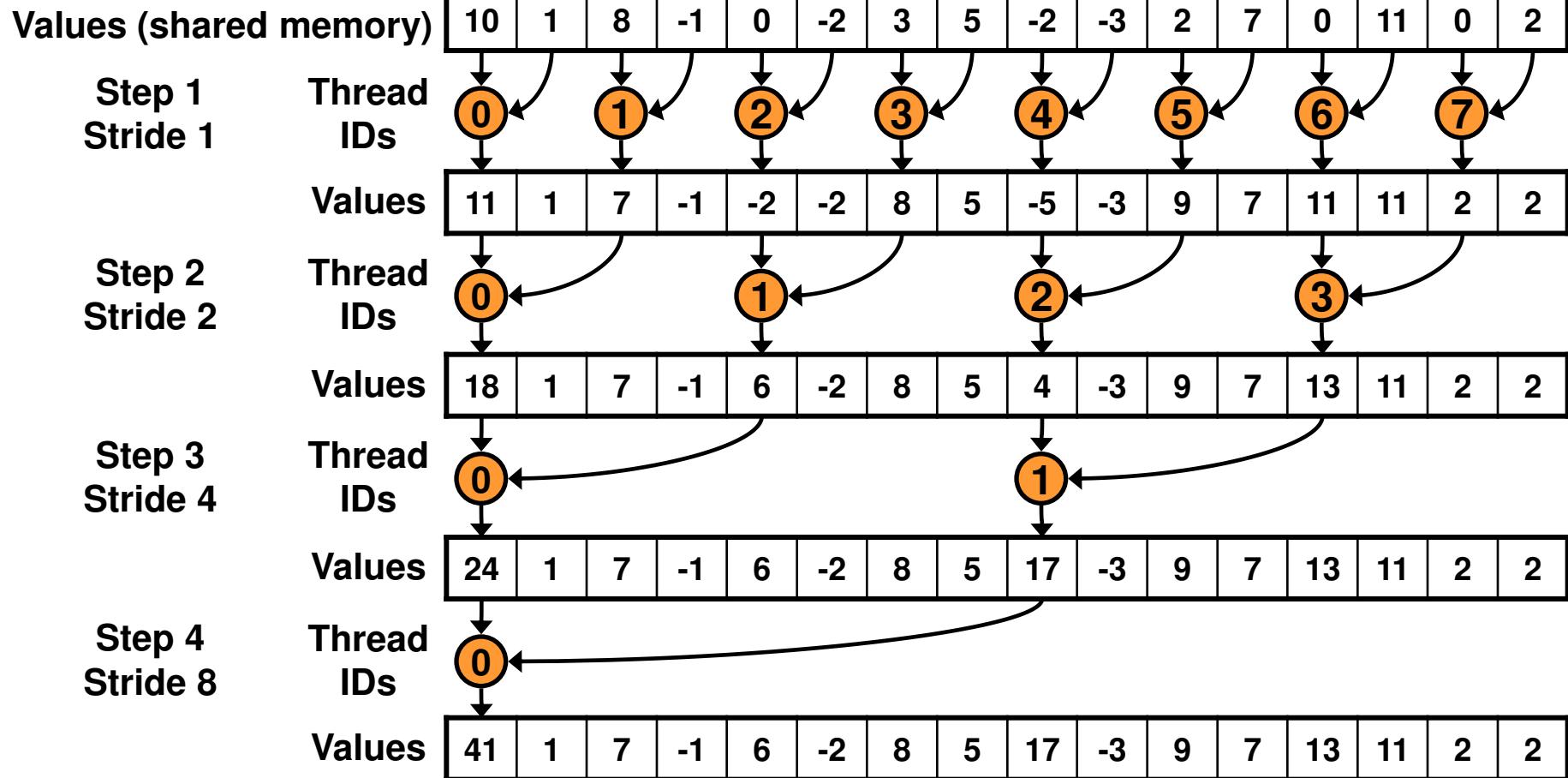
Just replace divergent branch in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {  
    if (tid % (2*s) == 0) {  
        sdata[tid] += sdata[tid + s];  
    }  
    __syncthreads();  
}
```

With strided index and non-divergent branch:

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {  
    int index = 2 * s * tid;  
  
    if (index < blockDim.x) {  
        sdata[index] += sdata[index + s];  
    }  
    __syncthreads();  
}
```

# Parallel Reduction: Interleaved Addressing



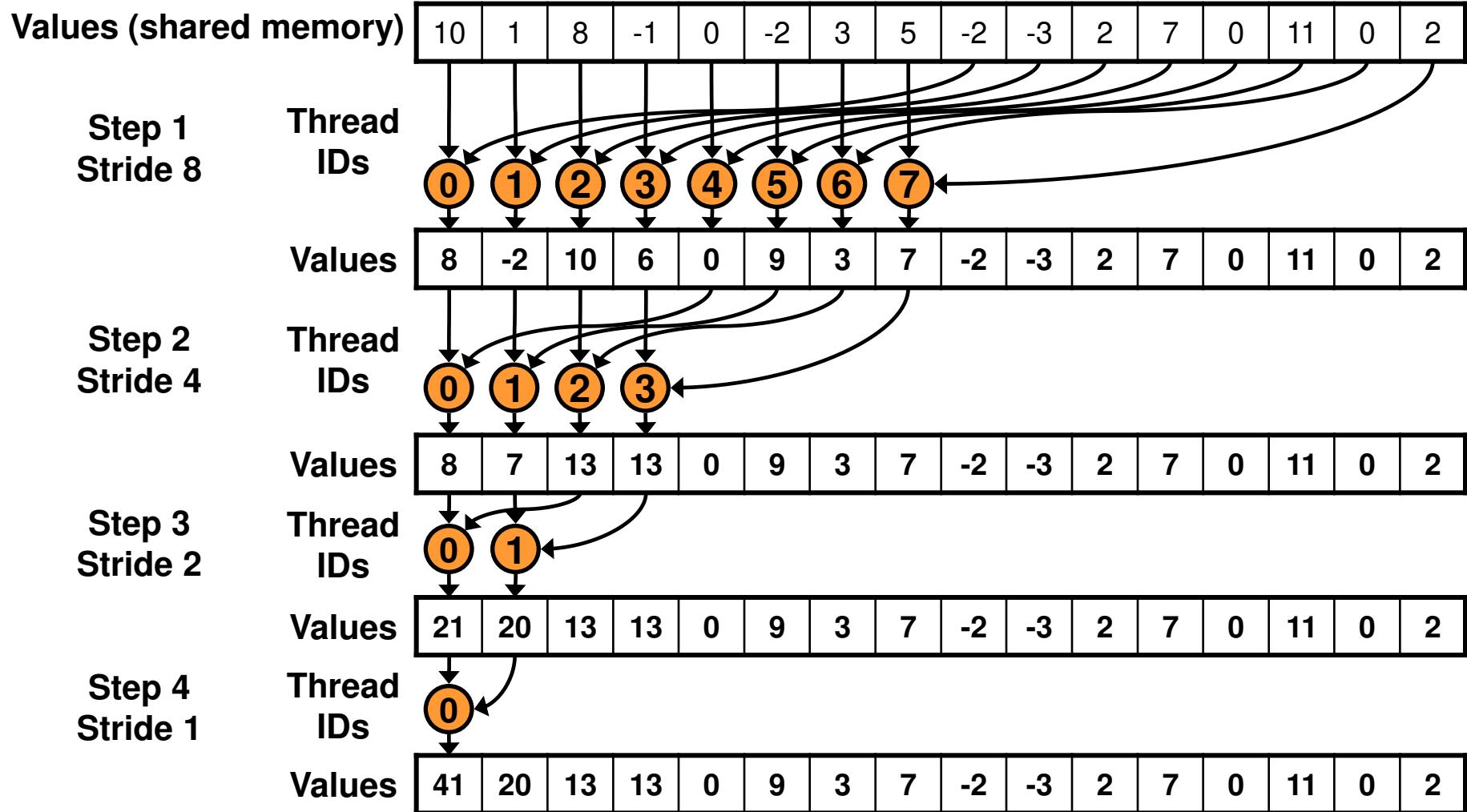
New Problem: Shared Memory Bank Conflicts

# Performance for 4M element reduction



	Time ( $2^{22}$ ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>

# Parallel Reduction: Sequential Addressing



Sequential addressing is conflict free

# Reduction #3: Sequential Addressing

Just replace strided indexing in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {  
    int index = 2 * s * tid;  
  
    if (index < blockDim.x) {  
        sdata[index] += sdata[index + s];  
    }  
    __syncthreads();  
}
```

With reversed loop and threadID-based indexing:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {  
    if (tid < s) {  
        sdata[tid] += sdata[tid + s];  
    }  
    __syncthreads();  
}
```

# Performance for 4M element reduction



	Time ( $2^{22}$ ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>	<b>4.68x</b>

# Idle Threads

Problem:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {  
    if (tid < s) {  
        sdata[tid] += sdata[tid + s];  
    }  
    __syncthreads();  
}
```

Half of the threads are idle on first loop iteration!

This is wasteful...



# Reduction #4: First Add During Load

Halve the number of blocks, and replace single load:

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

With two loads and first add of the reduction:

```
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

# Performance for 4M element reduction



	Time ( $2^{22}$ ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>	<b>4.68x</b>
<b>Kernel 4:</b> first add during global load	<b>0.965 ms</b>	<b>17.377 GB/s</b>	<b>1.78x</b>	<b>8.34x</b>

# Instruction Bottleneck



- ➊ At 17 GB/s, we're far from bandwidth bound
  - ➊ And we know reduction has low arithmetic intensity
- ➋ Therefore a likely bottleneck is instruction overhead
  - ➊ Ancillary instructions that are not loads, stores, or arithmetic for the core computation
  - ➋ In other words: address arithmetic and loop overhead
- ➌ Strategy: unroll loops

# Unrolling the Last Warp

- ➊ As reduction proceeds, # “active” threads decreases
  - ➌ When  $s \leq 32$ , we have only one warp left
- ➋ Instructions are SIMD synchronous within a warp
- ➌ That means when  $s \leq 32$ :
  - ➌ We don’t need to `__syncthreads()`
  - ➌ We don’t need “if ( $tid < s$ )” because it doesn’t save any work
- ➍ Let’s unroll the last 6 iterations of the inner loop



# Reduction #5: Unroll the Last Warp

```
__device__ void warpReduce(volatile int* sdata, int tid) {  
    sdata[tid] += sdata[tid + 32];  
    sdata[tid] += sdata[tid + 16];  
    sdata[tid] += sdata[tid + 8];  
    sdata[tid] += sdata[tid + 4];  
    sdata[tid] += sdata[tid + 2];  
    sdata[tid] += sdata[tid + 1];  
}
```

IMPORTANT:  
For this to be correct,  
we must use the  
“volatile” keyword!

```
// later...  
for (unsigned int s=blockDim.x/2; s>32; s>>=1) {  
    if (tid < s)  
        sdata[tid] += sdata[tid + s];  
    __syncthreads();  
}  
  
if (tid < 32) warpReduce(sdata, tid);
```

**Note: This saves useless work in *all* warps, not just the last one!**

Without unrolling, all warps execute every iteration of the for loop and if statement

# Performance for 4M element reduction



	Time ( $2^{22}$ ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>	<b>4.68x</b>
<b>Kernel 4:</b> first add during global load	<b>0.965 ms</b>	<b>17.377 GB/s</b>	<b>1.78x</b>	<b>8.34x</b>
<b>Kernel 5:</b> unroll last warp	<b>0.536 ms</b>	<b>31.289 GB/s</b>	<b>1.8x</b>	<b>15.01x</b>

# Complete Unrolling

- If we knew the number of iterations at compile time, we could completely unroll the reduction
  - Luckily, the block size is limited by the GPU to 512 threads
  - Also, we are sticking to power-of-2 block sizes
- So we can easily unroll for a fixed block size
  - But we need to be generic – how can we unroll for block sizes that we don't know at compile time?
- Templates to the rescue!
  - CUDA supports C++ template parameters on device and host functions

# Unrolling with Templates



- Specify block size as a function template parameter:

```
template <unsigned int blockSize>
__global__ void reduce5(int *g_idata, int *g_odata)
```

# Reduction #6: Completely Unrolled

Template <unsigned int blockSize>

```
__device__ void warpReduce(volatile int* sdata, int tid) {  
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];  
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];  
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];  
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];  
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];  
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];  
}
```

```
if (blockSize >= 512) {  
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }  
if (blockSize >= 256) {  
    if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }  
if (blockSize >= 128) {  
    if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads(); }  
  
if (tid < 32) warpReduce<blockSize>(sdata, tid);
```

Note: all code in RED will be evaluated at compile time.

Results in a very efficient inner loop!

# Invoking Template Kernels

- Don't we still need block size at compile time?

- Nope, just a switch statement for 10 possible block sizes:

```
switch (threads)
{
    case 512:
        reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 256:
        reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 128:
        reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 64:
        reduce5< 64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 32:
        reduce5< 32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 16:
        reduce5< 16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 8:
        reduce5< 8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 4:
        reduce5< 4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 2:
        reduce5< 2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 1:
        reduce5< 1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
}
```

# Performance for 4M element reduction



	Time ( $2^{22}$ ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>	<b>4.68x</b>
<b>Kernel 4:</b> first add during global load	<b>0.965 ms</b>	<b>17.377 GB/s</b>	<b>1.78x</b>	<b>8.34x</b>
<b>Kernel 5:</b> unroll last warp	<b>0.536 ms</b>	<b>31.289 GB/s</b>	<b>1.8x</b>	<b>15.01x</b>
<b>Kernel 6:</b> completely unrolled	<b>0.381 ms</b>	<b>43.996 GB/s</b>	<b>1.41x</b>	<b>21.16x</b>

# Parallel Reduction Complexity

- ➊ **Log( $N$ ) parallel steps, each step  $S$  does  $N/2^S$  independent ops**
  - ➌ **Step Complexity** is  $O(\log N)$
- ➋ **For  $N=2^D$ , performs  $\sum_{S \in [1..D]} 2^{D-S} = N-1$  operations**
  - ➌ **Work Complexity** is  $O(N)$  – It is **work-efficient**
  - ➌ i.e. does not perform more operations than a sequential algorithm
- ➌ With  $P$  threads physically in parallel ( $P$  processors), **time complexity** is  $O(N/P + \log N)$ 
  - ➌ Compare to  $O(N)$  for sequential reduction
  - ➌ In a thread block,  $N=P$ , so  **$O(\log N)$**

# What About Cost?

- ➊ **Cost of a parallel algorithm is processors time complexity**
  - ➌ Allocate threads instead of processors:  $O(N)$  threads
  - ➌ Time complexity is  $O(\log N)$ , so *cost* is  $O(N \log N)$  : **not cost efficient!**
- ➋ **Brent's theorem suggests  $O(N/\log N)$  threads**
  - ➌ Each thread does  $O(\log N)$  sequential work
  - ➌ Then all  $O(N/\log N)$  threads cooperate for  $O(\log N)$  steps
  - ➌ Cost =  $O((N/\log N) * \log N) = O(N) \rightarrow$  cost efficient
- ➌ **Sometimes called *algorithm cascading***
  - ➌ Can lead to significant speedups in practice

# Algorithm Cascading

- Combine sequential and parallel reduction
  - Each thread loads and sums multiple elements into shared memory
  - Tree-based reduction in shared memory
- Brent's theorem says each thread should sum  $O(\log n)$  elements
  - i.e. 1024 or 2048 elements per block vs. 256
- In my experience, beneficial to push it even further
  - Possibly better latency hiding with more work per thread
  - More threads per block reduces levels in tree of recursive kernel invocations
  - High kernel launch overhead in last levels with few blocks
- On G80, best perf with 64-256 blocks of 128 threads
  - 1024-4096 elements per *thread*



# Reduction #7: Multiple Adds / Thread

Replace load and add of two elements:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

With a while loop to add as many as necessary:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;

while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
}
__syncthreads();
```

# Reduction #7: Multiple Adds / Thread



Replace load and add of two elements:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

With a while loop to add as many as necessary:

```
unsigned int tid = th
unsigned int i = blo
unsigned int gridSize;
sdata[tid] = 0;

while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+gridSize];
    i += gridSize;
}
__syncthreads();
```

Note: gridSize loop stride  
to maintain coalescing!

# Performance for 4M element reduction



	Time ( $2^{22}$ ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>	<b>4.68x</b>
<b>Kernel 4:</b> first add during global load	<b>0.965 ms</b>	<b>17.377 GB/s</b>	<b>1.78x</b>	<b>8.34x</b>
<b>Kernel 5:</b> unroll last warp	<b>0.536 ms</b>	<b>31.289 GB/s</b>	<b>1.8x</b>	<b>15.01x</b>
<b>Kernel 6:</b> completely unrolled	<b>0.381 ms</b>	<b>43.996 GB/s</b>	<b>1.41x</b>	<b>21.16x</b>
<b>Kernel 7:</b> multiple elements per thread	<b>0.268 ms</b>	<b>62.671 GB/s</b>	<b>1.42x</b>	<b>30.04x</b>

Kernel 7 on 32M elements: 73 GB/s!



```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}

template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n) {
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

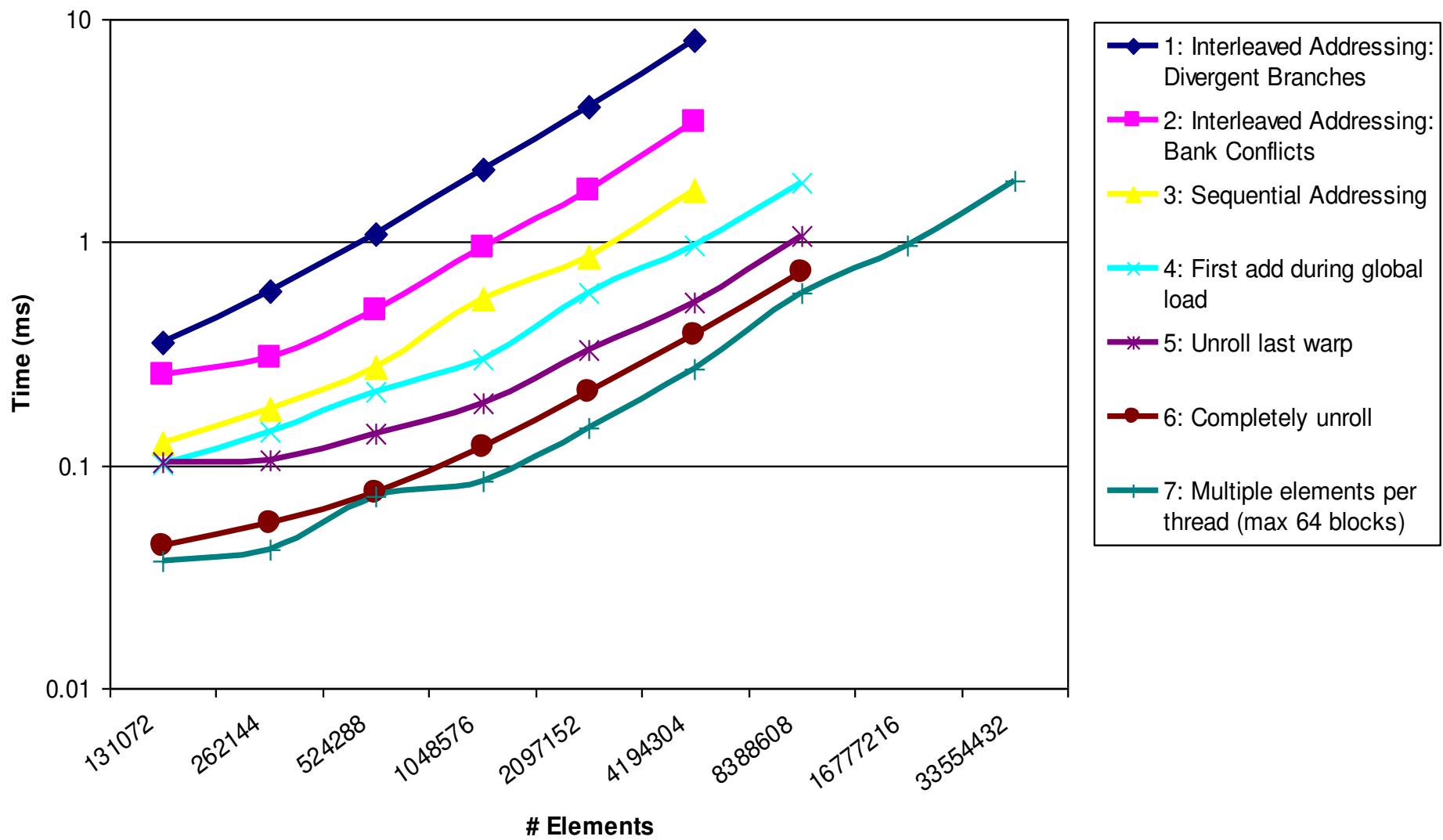
    while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize]; i += gridSize; }
    __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads(); }

    if (tid < 32) warpReduce(sdata, tid);
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

## Final Optimized Kernel

# Performance Comparison



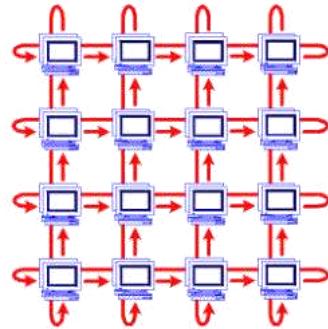


# Types of optimization

- ➊ Interesting observation:
- ➋ Algorithmic optimizations
  - ➌ Changes to addressing, algorithm cascading
  - ➌ 11.84x speedup, combined!
- ⌋ Code optimizations
  - ➌ Loop unrolling
  - ➌ 2.54x speedup, combined

# Conclusion

- ➊ Understand CUDA performance characteristics
  - ➊ Memory coalescing
  - ➋ Divergent branching
  - ➌ Bank conflicts
  - ➍ Latency hiding
- ➋ Use peak performance metrics to guide optimization
- ➌ Understand parallel algorithm complexity theory
- ➍ Know how to identify type of bottleneck
  - ➊ e.g. memory, core computation, or instruction overhead
- ➎ Optimize your algorithm, *then* unroll loops
- ➏ Use template parameters to generate optimal code
- ➐ Questions: [mharris@nvidia.com](mailto:mharris@nvidia.com)



---

# Parallel Computing

Tobias Lauer

Hochschule Offenburg

---

# Overview

---

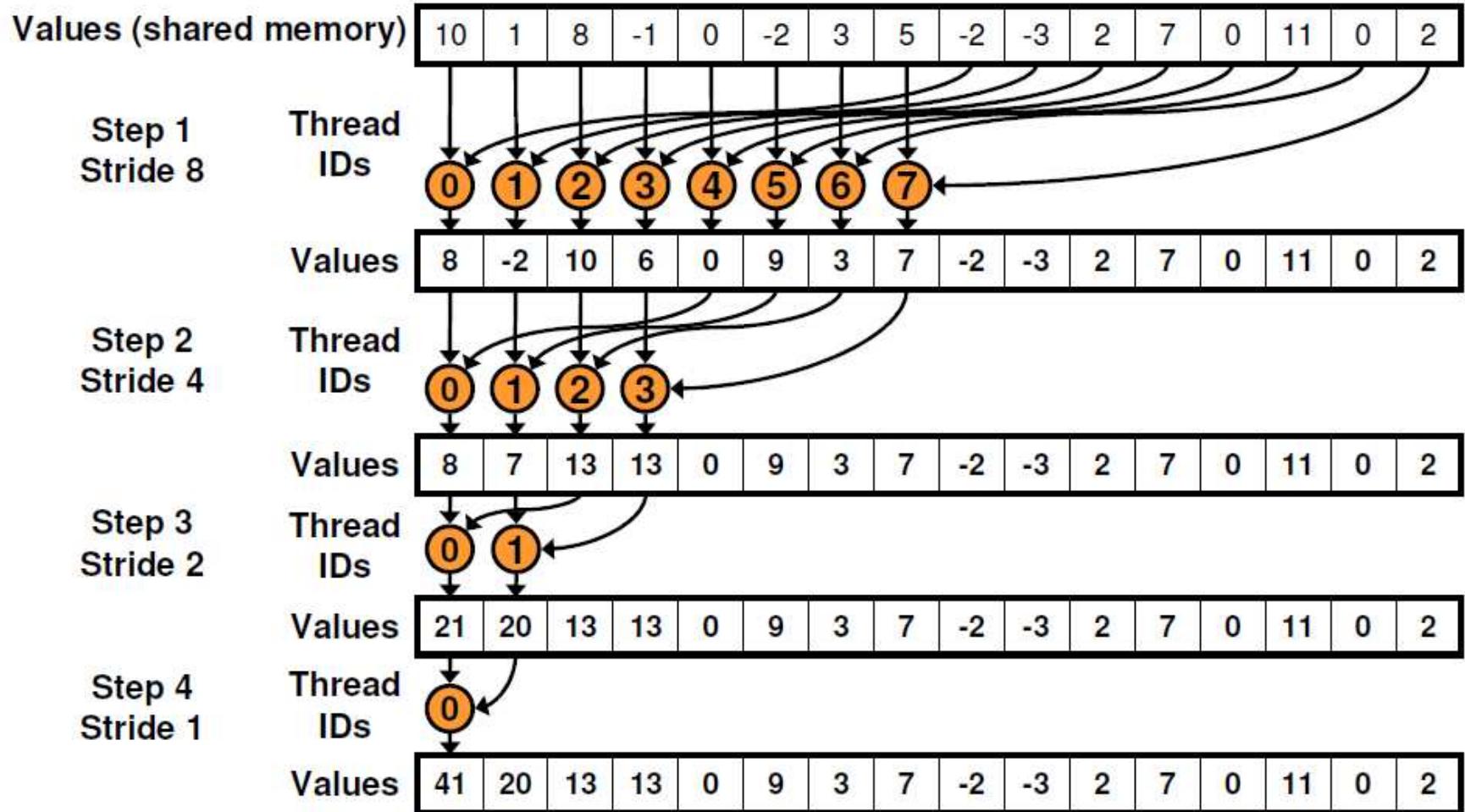
- CUDA-optimized algorithms
  - *Warp-internal functions for optimization*
  - *Atomic operations on recent CUDA architectures*

# Example: Reduction (Redux)

---

- Aggregate an input of  $n$  values to one single output value
  - *E.g. sum, min, max, product, count, ...*
- Sequential algorithm:  $O(n)$  time
- Parallel algorithm with tree-like structure
  - $O(\log n)$  time with  $O(n)$  processors
    - Can be improved to  $O(n / \log n)$  processors  
→ Cost complexity  $O(n)$  → cost-optimal
  - Requires synchronization between each level of the tree
    - In CUDA: `__syncthreads()`

# Parallel reduction algorithm



# Reduction

---

- Lots of potential for optimization (see Slides by Nvidia)
  - *Shared memory*
  - *Avoiding memory bank conflicts*
  - *Sequential computation per thread (algorithm cascading)*
  - ...
- Today: use SIMD characteristics of CUDA
  - *Direct access to registers of neighboring threads*
  - *No synchronisation required inside a warp*
- Also: use atomic operations
  - *Very efficient on recent CUDA architectures (not on older ones)*

# Recap: Warps

---

- A *warp* is a bundle of 32 „neighboring“ threads within a thread-block
  - *All threads operate in lockstep („im Gleichtakt“), i.e. they carry out the same instruction at the same clock cycle*
    - Exception: threads that don't have anything to do (because of a branch) do nothing
    - SIMT (single instruction multiple thread), „almost“ SIMD
  - *Inside the same warp, the thread IDs % 32 are 0..31*
    - i.e. for the first thread of a warp, `threadId.x % 32 == 0`

# Warp-internal operations

---

- Threads of the same warp have access to registers of other threads **in the same warp**
  - *Can use other threads' values of variables, share or broadcast their own values, ...*
- How can this be done (from a language point of view)?
  - *Problem: variable names are the same in all threads*
  - *Solution: Special warp-internal commands (intrinsics)*
    - `__ballot()`
    - `__shfl()`
    - ...

# The shuffle command (SHFL)

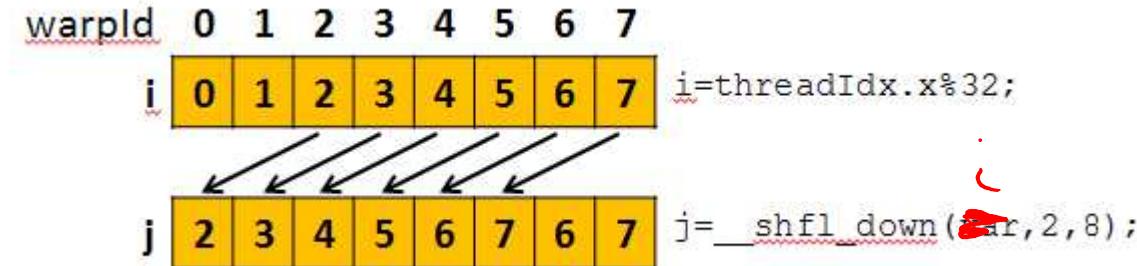
---

- `__shfl(int var, int sourceThread, int width)`
- `__shfl_down(int var, int threadDelta, int width)`
- `__shfl_up(int var, int threadDelta, int width)`
- `__shfl_xor(int var, int bitMask, int width)`

Returns the value of `var` of the thread in the same (part of the) warp (of size `width`), whose ID is given by the second parameter.

# Example: Shuffle-down

```
int i = threadIdx.x % 32;  
int j = __shfl_down(i, 2, 8);  
i = i + __shfl_down(i, 2, 8);
```



Source: <https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/>

# Advantages

---

- Sequence of several shared memory instructions is replaced by **one single instruction**
  - *Increases the bandwidth*
  - *Reduces latency*
- Requires **no** shared memory
  - *can be used for other things (limited resource)*
- Synchronization in warp is implicit in every instruction, hence **no need for \_\_syncthreads ()**
  - *Less block-wide synchronization required, which otherwise would decrease performance*

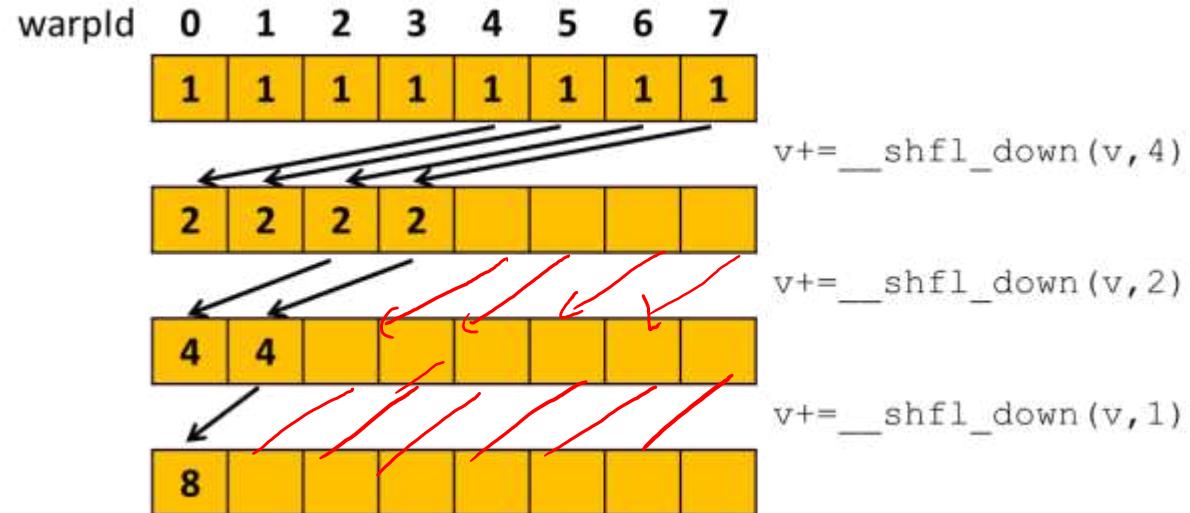
# Optimization of the reduction algorithm

---

- Idea:
  - *Reduction per warp*
    - using shuffle
  - *Reduction of all warp results in a block*
    - again using shuffle
  - *Reduction of all block results of the kernels*
    - by calling a second kernel
  - *The above limits the number of blocks ( $\leq 1024 = 32*32$ )*
    - Solution: start with a sequential reduction per thread

# Warp reduction

```
__device__ int warpReduceSum(int val) {  
    for (int offset = warpSize/2; offset > 0; offset /= 2)  
        val += __shfl_down(val, offset);  
    return val;  
}
```



# Block reduction using warp reduction

```
__device__ int blockReduceSum(int val) {  
  
    static __shared__ int shared[32];           // Shared mem for 32 partial sums  
    int lane = threadIdx.x % warpSize;         // thread id inside warp  
    int wid  = threadIdx.x / warpSize;          // warp id inside block  
  
    val = warpReduceSum(val);                  // Each warp performs partial reduction  
  
    if (lane==0) shared[wid]=val;              // Write reduced value to shared memory  
  
    __syncthreads();                         // Wait for all partial reductions  
  
    //read from shared memory only if that warp existed  
    val = (threadIdx.x < blockDim.x / warpSize) ? shared[lane] : 0;  
  
    if (wid==0) val = warpReduceSum(val);      // Final reduce within first warp  
  
    return val;  
}
```

# Complete kernel

```
__global__ void deviceReduceKernel(int *in, int* out, int N) {  
  
    int sum = 0;  
  
    // sequentially pre-reduce data to size of grid  
    for (int i = blockIdx.x * blockDim.x + threadIdx.x;  
         i < N;  
         i += blockDim.x * gridDim.x) {  
        sum += in[i];  
    }  
  
    sum = blockReduceSum(sum);  
  
    if (threadIdx.x==0)          // first thread writes out block result  
        out[blockIdx.x]=sum;  
}
```

# Kernel call

---

```
void deviceReduce(int *in, int* out, int N) {  
    int threads = 512;          // enough to saturate GPU  
    int blocks  = min((N + threads - 1) / threads, 1024);  
  
    deviceReduceKernel<<<blocks, threads>>>(in, out, N);  
  
    // Second kernel call for final reduction  
    deviceReduceKernel<<<1, 1024>>>(out, out, blocks);  
}
```

# Further optimization?

---

Potentially inefficient:

- Second kernel call
  - *Under-occupies GPU (only one SM is used)*
  - *Global synchronization between kernels*
    - Need to wait for **every thread** in **every block**
- Block reduction uses **`__syncthreads()`**
  - *Block-wide synchronization*
    - Need to wait for **every thread** in **that block**

# Atomic Add

---

- Thread-safe addition of a value to a variable
  - *Implicitly synchronized – no manual synchronization required*
- Syntax: `atomicAdd(variable, valueToBeAdded);`
- Attention: this is a **blocking** operation!
  - *Only 1 thread per cycle can add*
  - *Threads in same warp will be serialized*
  - *Danger of contention of threads („Staugefahr“)*

# Previous kernel

```
__global__ void deviceReduceKernel(int *in, int* out, int N) {  
  
    int sum = 0;  
  
    for (int i = blockIdx.x * blockDim.x + threadIdx.x;  
         i < N; i += blockDim.x * gridDim.x) {  
        sum += in[i];  
    }  
  
    sum = blockReduceSum(sum);  
  
    if (threadIdx.x==0)          // first thread writes out  
        out[blockIdx.x]=sum;     // block result  
}
```

# New kernel

```
__global__ void deviceReduceBlockAtomicKernel(int *in, int* out,
                                              int N) {
    int sum = 0;

    for (int i = blockIdx.x * blockDim.x + threadIdx.x;
          i < N; i += blockDim.x * gridDim.x) {
        sum += in[i];
    }

    sum = blockReduceSum(sum);

    if (threadIdx.x==0)           // first thread atomically adds
        atomicAdd(out, sum);     // block result to final result
}
```

# What has changed?

---

- Don't need a second kernel call
  - *Each **block** directly adds its own partial result to final result*
- Only 1 thread **per block** will call **atomicAdd**
  - *Danger of contention is small*
- But: atomicAdd synchronizes implicitly
  - *Can this be more efficient?*
    - Yes! Throughput increased by up to 10%
    - But only works on recent CUDA architectures (Kepler and later)
    - Noticeable for  $N$  between 200.000 and 100.000.000

# Further optimization?

---

Potentially inefficient:

- Second kernel call
  - *Under-occupies GPU (only one SM is used)*
  - *Global synchronization between kernels*
    - Need to wait for **every thread** in **every block**
- Block reduction uses **`__syncthreads()`**
  - *Block-wide synchronization*
    - Need to wait for **every thread** in **that block**
- Idea: even more atomics instead of block reduction



# Previous kernel

```
__global__ void deviceReduceBlockAtomicKernel(int *in, int* out,
                                              int N) {
    int sum = 0;

    for (int i = blockIdx.x * blockDim.x + threadIdx.x;
          i < N; i += blockDim.x * gridDim.x) {
        sum += in[i];
    }

    sum = blockReduceSum(sum);

    if (threadIdx.x==0)           // first thread atomically adds
        atomicAdd(out, sum);     // block result to final result
}
```

# New kernel

```
__global__ void deviceReduceWarpAtomicKernel(int *in, int* out,
                                             int N) {
    int sum = 0;

    for (int i = blockIdx.x * blockDim.x + threadIdx.x;
          i < N; i += blockDim.x * gridDim.x) {
        sum += in[i];
    }

    sum = warpReduceSum(sum);

    if (threadIdx.x & (warpSize-1) == 0) // first thread in warp
        atomicAdd(out, sum); // adds warp result to final result
}
```

# What has changed?

---

- No block reduction at all!
  - *Each warp directly adds its own partial result to final result*
- 1 Thread per warp will call **atomicAdd**
  - *Higher danger of contention than before!*
- Still more efficient?
  - *Yes! Throughput increased by up to 50%*
    - But only works on recent CUDA architectures (Kepler and later)
    - Noticeable for  $N$  between 150.000 and 50.000.000

# Summary

---

- Warp internal functions can use SIMD-like features to avoid blockwide synchronization (**\_\_syncthreads**)
- Atomic operations
  - *Performance has been greatly improved in recent CUDA architectures!*
  - *Can increasingly be used als alternative to explicitly synchronized algorithms*



**Hochschule Offenburg**  
offenburg.university

# Accelerating Density-Based Subspace Clustering in High-Dimensional Data

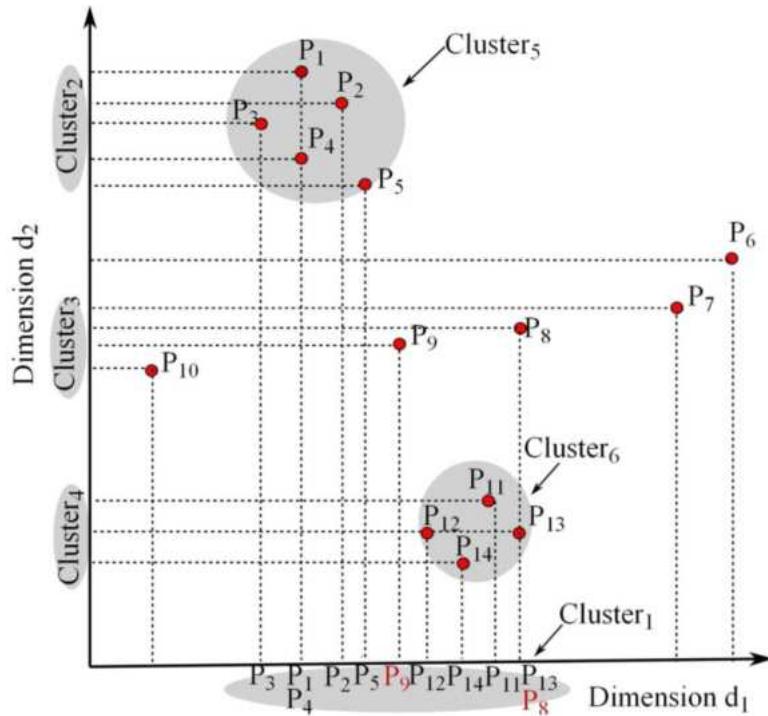
Jürgen Prinzbach, Tobias Lauer, Nicolas Kiefer

*To appear: Workshop on High-Dimensional Data Mining (HDM '21)*

im Rahmen der [IEEE International Conference on Data Mining \(ICDM '21\)](#)

# Data Clustering

- Finde (disjunkte) Teilmengen **ähnlicher Daten** in großen/komplexen Datenmengen
- Ähnlichkeit definiert über **Distanzfunktion**
  - Euklidische Distanz, Manhattan-Distanz, Hamming-Distanz, ...
- Grundlegende Ansätze:
  - Segmentiere den **Raum** (z.B. *k*-means)
  - Teile die Punkte **dichtebasiert** auf (z.B. DBSCAN)
    - Cluster = **Menge von  $\geq k$  Punkten mit Abstand  $< \varepsilon$**  ( $k, \varepsilon$  wählbare Parameter)
    - Unterstützt auch Ausreißer



# High-Dimensional Data

- Hohe Zahl an Attributen, evtl. größer als Zahl der Datenpunkte
  - (Globale) Distanzen verlieren Aussagekraft
    - Unterschiede in wenigen Dimensionen → hohe Distanz trotz „Ähnlichkeit“
    - Punkte stets weit voneinander entfernt; es gibt keinen „guten“ Wert für  $\varepsilon$
- Cluster werden **nicht gefunden**

Lösungsansätze:

- Dimensionsreduktion
- Hauptkomponentenanalyse (PCA)
- Deep Neural Networks
- **Subspace Clustering**

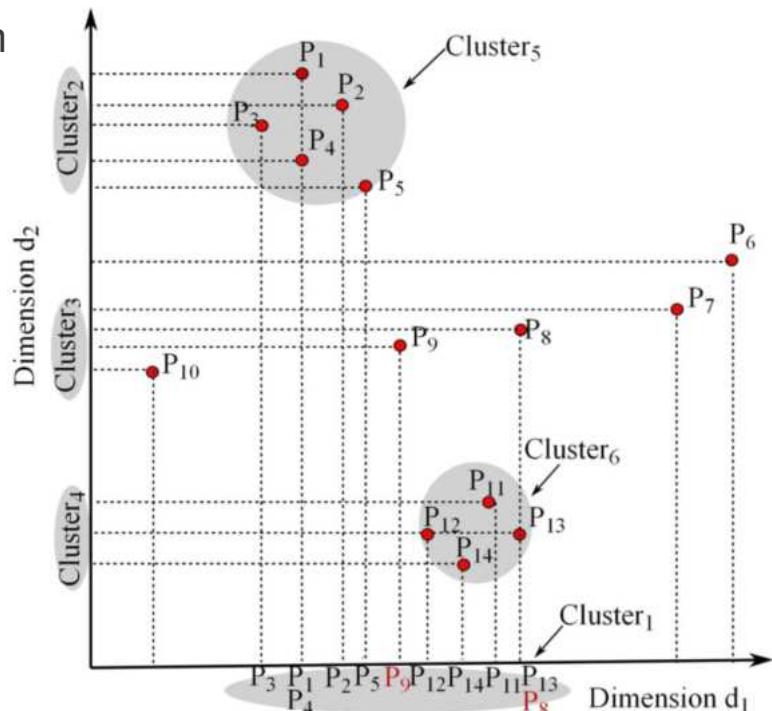
# Subspace Clustering

Suche Cluster in **Teilräumen** (Subspaces), d.h. in Räumen, die jeweils nur von einer **Teilmenge der Attribute** aufgespannt werden

- Problem:  $d$  Attribute  $\rightarrow 2^d$  Subspaces
  - Alle Subspaces zu prüfen ist oft unmöglich ( $2^{100} > 1$  Quadrilliarde)
  - Notwendig: Vermeidung redundanter Berechnungen
- Bekannte existierende Ansätze:  
**CLIQUE, MAFIA, SUBCLU, SUBSCALE**

# SUBSCALE (Kaur & Datta 2015)

- (1) Identifiziere Cluster in jedem 1-dimensionalen Subraum
- (2) Bestimme für jedes Cluster *C* alle Teilmengen der Größe  $k$  (*dense units* = *Teilcluster kleinster Größe*)
- (3) Prüfe, ob diese in anderen Dimensionen ebenfalls *dense units* darstellen → mittels *Hashkollision*: jede *dense unit* erhält eine eindeutige\* Signatur, die gehasht wird
- (4) Baue dadurch sukzessive höherdimensionale *dense units* auf, bis diese maximale Dimensionalität haben
- (5) Vereinige in jedem Subraum überlappende *dense units* zu Clustern maximaler Größe



\*eindeutig „mit sehr hoher Wahrscheinlichkeit“ (Erdös & Lehner, 1941)

# SUBSCALE (Kaur & Datta 2015)

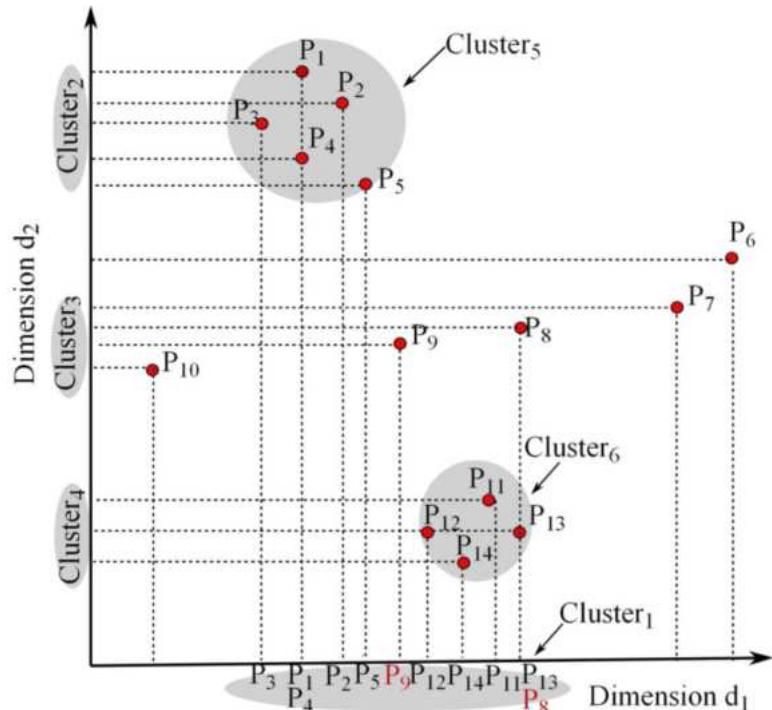
- (1) Identifiziere Cluster in jedem 1-dimensionalen Subraum
- (2) Bestimme für jedes Cluster  $C$  alle Teilmengen der Größe  $k$   
(*dense units* = *Cluster kleinster Größe*)
- (3) Prüfe, ob diese in anderen Dimensionen ebenfalls *dense units* sind
- (4) Baue dadurch sukzessive höherdimensionale *dense units* auf, bis diese maximale Dimensionalität haben
- (5) Vereinige in jedem Subraum überlappende *dense units* zu Clustern maximaler Größe

**Pro:** Nur 1 Durchgang pro Dimension:  $O(d)$  statt  $O(2^d)$  !

**Con:** Durchgänge (Schritt (2) & (3)) sind **rechenaufwendig**:  
Bestimmung aller Teilmengen der Größe  $k$   
Es gibt  $\binom{|C|}{k}$  solche Teilmengen pro Cluster  $C$   
→ **kombinatorische Explosion**

Laufzeit zwar nicht in  $O(2^d \cdot n)$ , aber in  $O(d \cdot n^k)$

Außerdem: Hoher Platzbedarf für Hashtabelle



# Beschleunigung durch Parallelisierung

Möglichkeiten:

- Paralleles Abarbeiten der Dimensionen
  - Bringt weniger als erhofft (Datta et al. 2017)
- Paralleles Abarbeiten von Teilen der Hashtabelle
  - Skaliert zwar linear mit #Prozessoren, aber bei höherer Gesamtarbeit  
(1 Datendurchgang pro Partition pro Dimension)  
→ Realer Zeitgewinn <50%
- Paralleles Abarbeiten einzelner *dense units*
  - Sehr feingranular
  - Viele (Millionen) kleine Tasks  
→ GPU als geeignete Hardware

# Beschleunigung durch Parallelisierung

- GPU: viele Prozessoren (>5000)
- Datenparallelität:  
GPU-Threads bearbeiten identische Task, aber auf unterschiedlichem Input („Single Instruction Multiple Thread“ – SIMD)
- Hier: Jeder GPU-Thread bearbeitet 1 *dense unit*:  
Thread *i*:
  - (1) Bestimme die *i*-te Teilmenge (*dense unit*)
  - (2) Berechne deren eindeutige Signatur
  - (3) Trage die Teilmenge anhand der Signatur in die Hashtabelle ein

# „Paralleles Aufzählen“ von Teilmengen

- Teilmengen (*dense units*) könnten auf einfache Weise durch Aufzählen in (co)lexikographischer Reihenfolge ermittelt werden.
  - Teilmenge  $i$  kann in  $O(k)$  bestimmt werden
  - Aber: für Teilmenge  $i$  muss Teilmenge  $i-1$  bereits bekannt sein
  - Sequenziell, nicht parallelisierbar
- Alternative:  
Bestimme jede Teilmenge unabhängig, nur anhand ihrer Nummer  $i$ 
  - Nachteil: Aufwendiger pro Teilmenge:  $O(k \cdot \log n)$
  - Aber: Teilmengen können parallel berechnet werden

# Binomiale Zerlegung einer Zahl $i$

- Theorem (vgl. Kruskal 1963, Katona 1968)

Jede natürliche Zahl  $i$  lässt sich für jedes  $k > 0$  eindeutig als Summe von genau  $k$  Binomialkoeffizienten darstellen (mit  $0 \leq n_1 < n_2 < \dots < n_k$ ):

$$i = \binom{n_1}{1} + \binom{n_2}{2} + \dots + \binom{n_k}{k}$$

Wir nennen dies die **binomiale** oder **kombinatorische Zerlegung** von  $i$ .

- Beobachtung:

$n_1, \dots, n_k$  entsprechen den Elementnummern der  $i$ -ten Teilmenge von  $C$

# Direkte Berechnung der $i$ -ten Teilmenge

Beispiel:  $i = 7$ ,  $k = 4$

$$7 = \binom{0}{1} + \binom{2}{2} + \binom{3}{3} + \binom{5}{4} = 0 + 1 + 1 + 5$$

Die Menge  $\{ 0, 2, 3, 5 \}$  ist gerade die 7. Teilmenge mit 4 Elementen in colexikographischer Aufzählung:

0: {0,1,2,3}  
1: {0,1,2,4}  
2: {0,1,3,4}  
3: {0,2,3,4}

4: {1,2,3,4}  
5: {0,1,2,5}  
6: {0,1,3,5}  
**7: {0,2,3,5}**

8: {1,2,3,5}  
9: {0,1,4,5}  
10: {0,2,4,5}  
11: {1,2,4,5}

# Datenparalleler Algorithmus

Berechnet für ein 1-dimensionales Cluster  $C = \{ p_1, p_2, \dots, p_N \}$  in Dimension  $d$  alle *dense units* und trägt diese in die Hashtabelle  $H$  ein:

```
1  for i = 0 to  $\binom{N}{k}$ -1 in parallel
2      find  $n_1, \dots, n_k$  such that  $i = \sum_{j=1}^k \binom{n_j}{j}$ 
3       $D_i = \{ p_{n_1}, \dots, p_{n_k} \}$ 
4      calculate signature  $S(D_i)$ 
5      if  $H$  contains  $S(D_i)$ 
6          append  $d$  to dimension list of element  $S(D_i)$ 
7      else
8          insert new element  $S(D_i)$  with  $d$  in dimension list
```

# Implementation und Tests

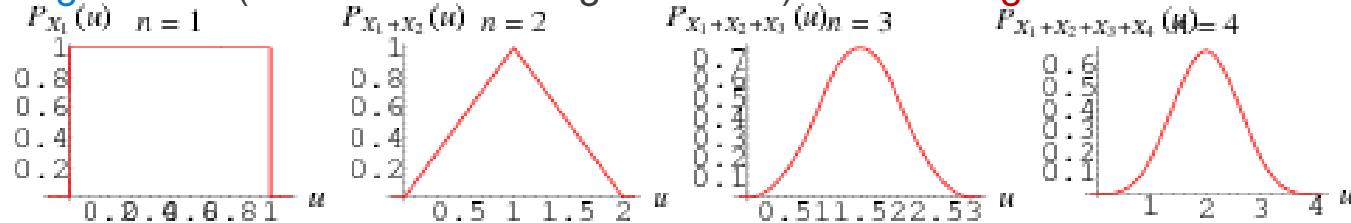
---

- Master-Thesis (Nicolas Kiefer, INFM)
  - Implementation für NVIDIA GPUs mit CUDA/C++
  - Effizientes GPU-Hashverfahren „Stadium Hashing“ (Khorasani et al., 2015)
  - Vergleichstests mit bisherigen Verfahren
- Ergebnisse
  - GPU-Algorithmus in SUBSCALE-Verfahren integriert
  - Beschleunigung um ca. Faktor 5 gegenüber bisheriger Version

- Große Datenmengen  
→ Hashtabelle zu groß für RAM
- Einfache Lösung: **Partitionierung** der Hashtabelle (nach Hashwert = Signatur)
  - Iteriere über alle  $p$  Partitionen
  - Führe **für jede Partition** den Algorithmus aus
  - Signatur nur hashen, wenn sie in aktuelle Partition fällt

# Signaturen und Hashtable-Partitionen

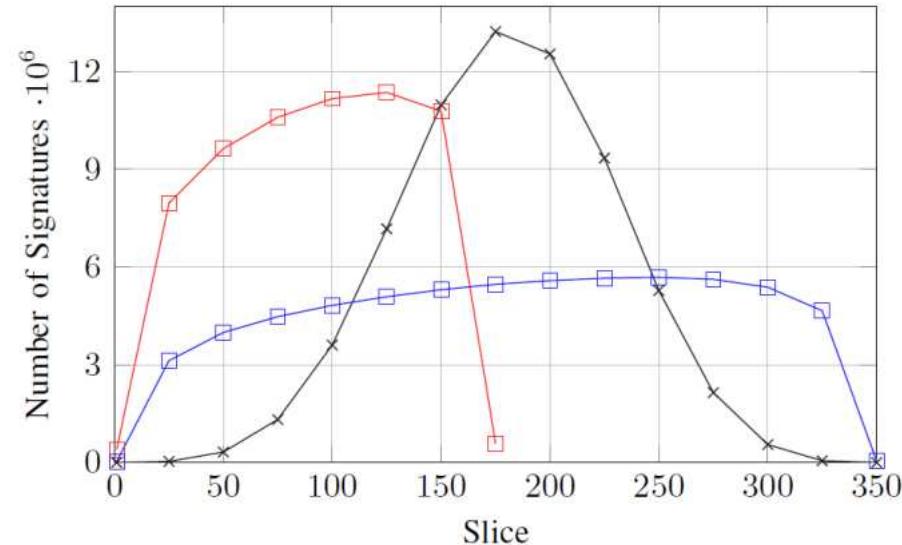
- SUBSCALE teilt den Bereich der Hashwerte in **gleich breite Partitionen**
- Aber **Signaturen** (= Summen zufälliger Zahlen) sind **nicht gleichverteilt**



- Großteil der Partitionen wird **nur gering befüllt** → ineffizient

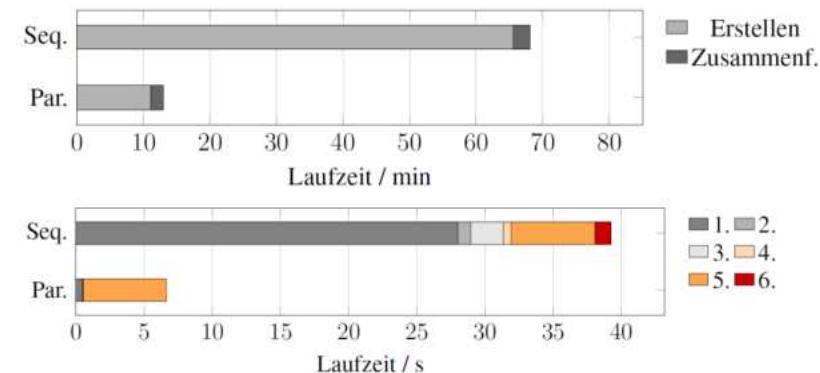
# Weitere Optimierung

- Anpassung der Partitionsgrößen an die Verteilungsfunktion
- Ca. 50% Einsparung an Speicherplatz oder an Partitionen
- Ca. 40% Zeitersparnis für gesamtes Clustering



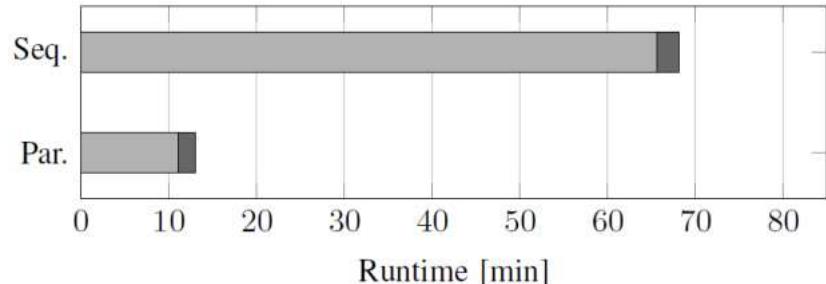
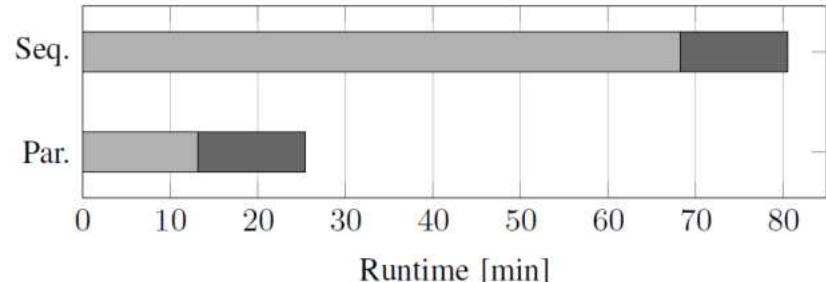
# Implementation und Tests

- Tests
  - GPU-Algorithmus in SUBSCALE-Verfahren integriert
  - Beschleunigung um ca. Faktor 5 gegenüber bisheriger Version

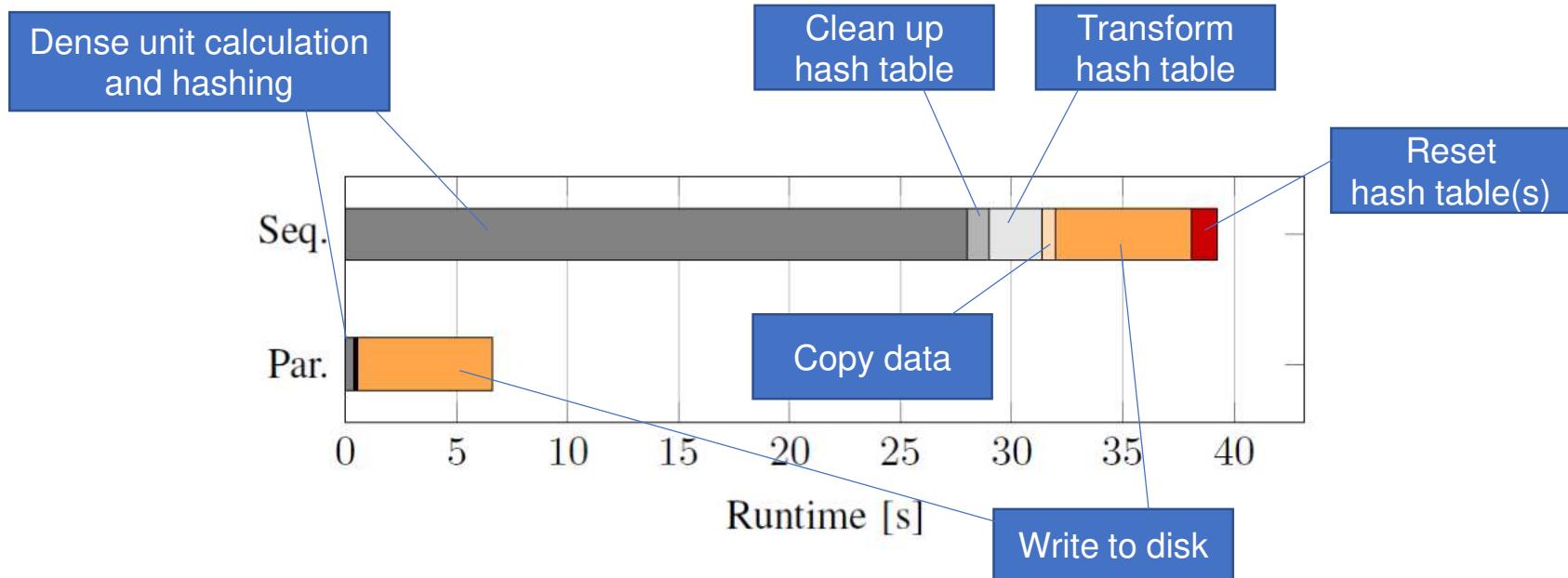


# Performance Tests

- Overall Runtime
  - SUBSCALE
  - DBSCAN (merge dense units)
- SUBSCALE
  - Partitionen berechnen
  - Partitionen verbinden



# Berechnung einer Partition – Zeitanteile



- Speedup calculation+hashing: ~70x

---

# GPU-accelerated Weighted Aggregation and Disaggregation in Multidimensional Databases

---

Tobias Lauer

Hochschule Offenburg

# Business Intelligence and Corporate Planning

**Berichts-Manager**

**Status Monitor**

**BikersBest**

Year	2011	Region	East	Quarter	Qtr.1
Austria	Sales	911.400	947.856	976.292	
	Units	3.557	20.553	21.170	
Czech Republic	Sales	722.367	751.261	773.799	
	Units	3.365	9.535	9.821	
Poland	Sales	692.889	720.604	742.222	
	Units	5.029	7.474	7.698	
Slovakia	Sales	606.812	631.082	650.014	
	Units	2.553	2.818	2.903	
Slovenia	Sales	0	0	0	
	Units	0	0	0	

**Berichts-Manager**

**TopDown Planung**

**BikersBest**

**Budget Planning - Sales - Top Down - 2011**

25.04.2012

Budget Allocation for 2011

Base year data

Actual

2010

select a default Budget allowance

Actual 2010 + Manual Adjustment

5%

59.553.952

Copy to Budget

Actual

Budget

Details

2007      2008      2009      2010      Δ Act.10/Bud.11      2011

**East**

	2007	Δ 07/08	2008	Δ 08/09	2009	Δ 09/10	2010	Δ Act.10/Bud.11	2011
Austria	10.277.846	▲ 8,4%	11.137.088	▼ -3,4%	10.761.185	▼ -6,6%	10.025.043	▲ 5,0%	10.526.295
Czech Republic	3.050.189	▲ 7,4%	3.276.984	▼ -4,2%	3.139.532	▼ -6,6%	2.933.240	▲ 5,0%	3.079.902
Poland	2.463.432	▲ 7,5%	2.647.448	▼ -4,2%	2.536.103	▼ -6,6%	2.369.379	▲ 5,0%	2.487.848
Slovakia	2.793.216	▲ 10,1%	3.075.061	▼ -1,6%	3.026.918	▼ -7,4%	2.803.823	▲ 5,0%	2.944.014
Slovenia	1.971.009	▲ 8,5%	2.137.594	▼ -3,7%	2.058.631	▼ -6,8%	1.918.600	▲ 5,0%	2.014.530

**North**

	2007	Δ 07/08	2008	Δ 08/09	2009	Δ 09/10	2010	Δ Act.10/Bud.11	2011
Denmark	9.128.986	▲ 7,6%	9.818.834	▼ -0,4%	9.777.665	▼ -4,8%	9.311.918	▲ 5,0%	9.777.514
Finland	1.996.506	▲ 7,7%	2.150.407	▼ -0,3%	2.142.901	▼ -4,8%	2.039.352	▲ 5,0%	2.141.319
Norway	2.411.643	▲ 7,6%	2.594.918	▼ -0,4%	2.584.087	▼ -4,8%	2.460.646	▲ 5,0%	2.583.678
Sweden	1.644.309	▲ 7,4%	1.766.463	▼ -0,4%	1.758.860	▼ -4,7%	1.675.824	▲ 5,0%	1.759.616

**South**

	2007	Δ 07/08	2008	Δ 08/09	2009	Δ 09/10	2010	Δ Act.10/Bud.11	2011
Portugal	9.771.582	▲ 8,0%	10.549.642	▼ -3,0%	10.233.229	▼ -9,3%	9.276.810	▲ 5,0%	9.740.650
Spain	29.329.454	▲ 8,8%	31.916.952	▼ -5,2%	30.261.612	▼ -7,1%	28.104.278	▲ 5,0%	29.509.492

**West**

	2007	Δ 07/08	2008	Δ 08/09	2009	Δ 09/10	2010	Δ Act.10/Bud.11	2011
Netherlands	1.811.221	▲ 10,0%	1.992.340	▲ 10,0%	1.911.111	▲ 10,0%	1.831.134	▲ 10,0%	1.831.134
Switzerland	1.831.134	▲ 10,0%	2.014.257	▲ 10,0%	2.014.257	▲ 10,0%	2.014.257	▲ 10,0%	2.014.257
United Kingdom	7.447.307	▲ 10,0%	8.192.027	▲ 10,0%	8.192.027	▲ 10,0%	8.192.027	▲ 10,0%	8.192.027
Total	56.718.049	▲ 10,0%	68.152.705	▲ 10,0%	68.152.705	▲ 10,0%	68.152.705	▲ 10,0%	68.152.705

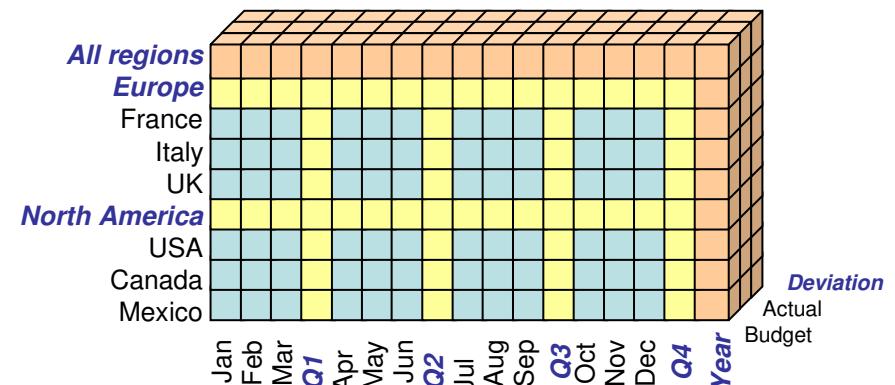
**All Customers**

	2007	Δ 07/08	2008	Δ 08/09	2009	Δ 09/10	2010	Δ Act.10/Bud.11	2011
All Customers	58.507.868	▲ 8,4%	63.422.515	▼ -3,8%	61.033.691	▼ -7,1%	56.718.049	▲ 5,0%	59.553.952

GEO

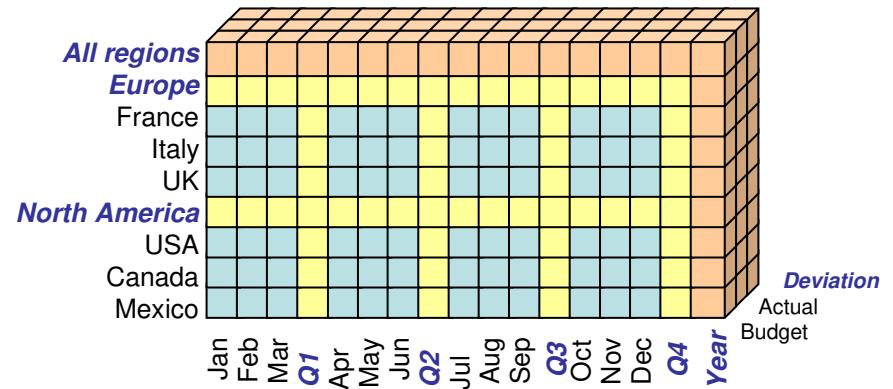
# Online Analytical Processing (OLAP)

- Data modeled as multidimensional “cube”
  - *Dimensions are structured hierarchically:*
    - Base elements
    - Consolidated elements
  - *Data cube operations:*
    - Slice, dice, pivot across dimensions
    - Roll-up, drill-down along hierarchies



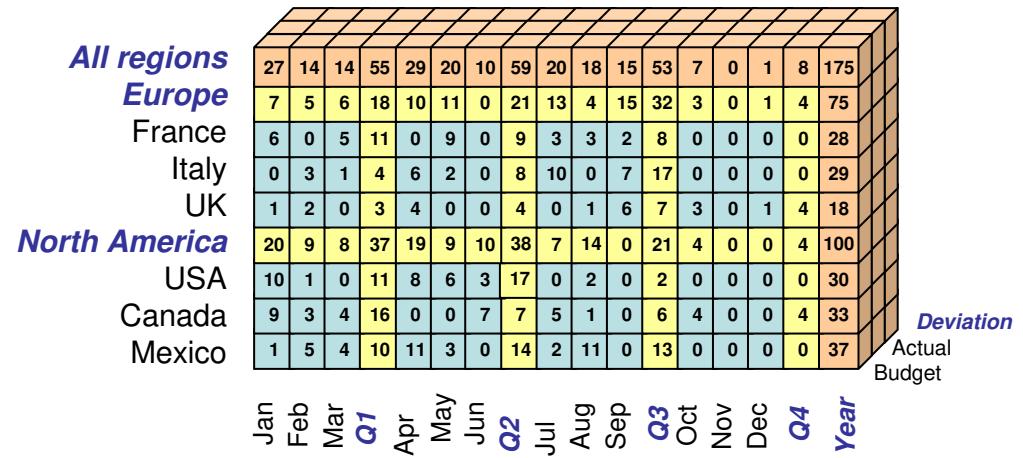
# Multidimensional aggregation

- Pre-computation of aggregates
  - *not possible for all aggregates (dimensional explosion)*
  - *not desired*
- Online aggregation
  - *On demand*
  - *Useful for planning scenarios (interactive write-back)*
  - *Common when data is stored in-memory*



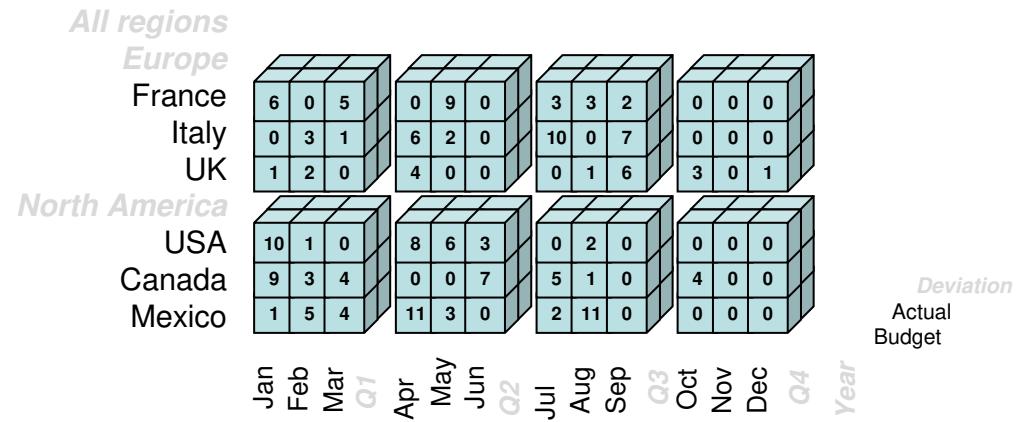
# In-memory OLAP storage model

- All data stored in main memory



# In-memory OLAP storage model

- All data stored in main memory
  - Only store base cells



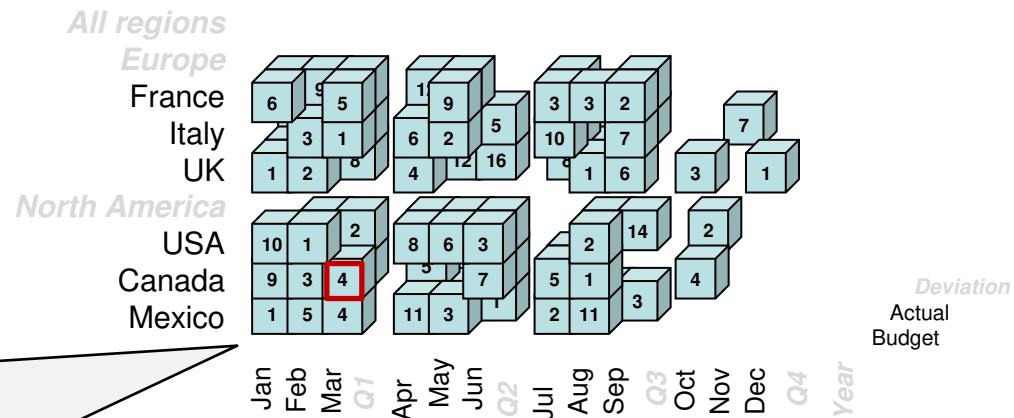
# In-memory OLAP storage model

- All data stored in main memory
- Only store base cells
- Do not store zero-value cells

→ Memory saving, data consistency

Represent cells as (key, value) pairs,  
e.g.  
 $( (2, 1, 0), 4.0 )$

Note: Values are double precision!



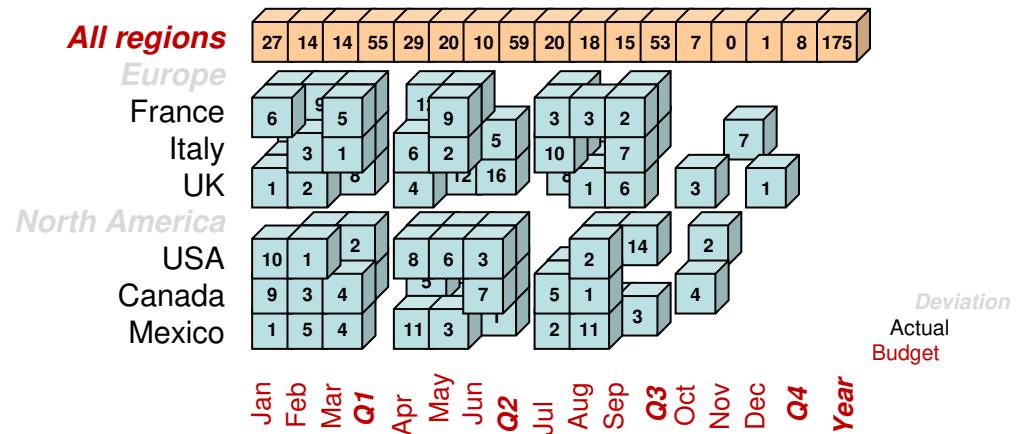
# In-memory OLAP storage model

- All data stored in **GPU** memory
- Only store base cells
- Do not store zero-value cells

→ Memory saving, data consistency

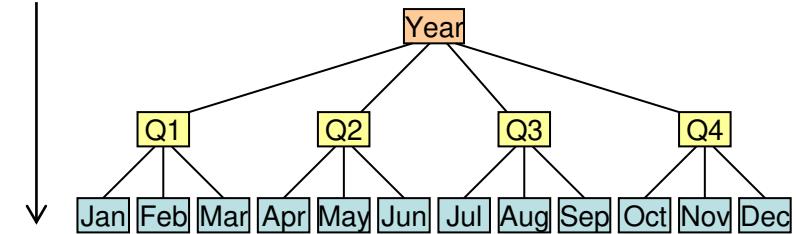
- Compute other cells *on the fly* when needed

→ Use GPU to accelerate

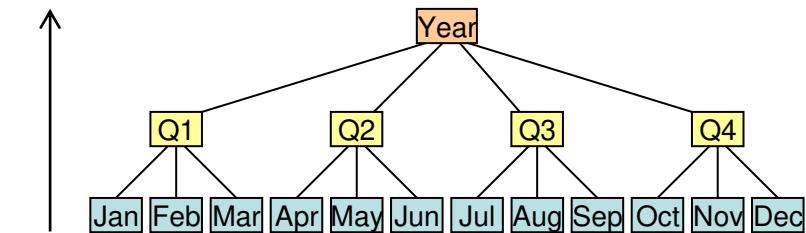


# Two algorithmic approaches to GPU aggregation

- Target-driven
  - *For each target cell*
    - find relevant base cells (parent-to-child map)
    - aggregate



- Source-driven
  - *Pre-filter base cells relevant for area of all target cells*
  - *For each base cell*
    - create all relevant target paths (child-to-parent map)
    - look up in hashmap
    - add to aggregate



# Target-driven aggregation

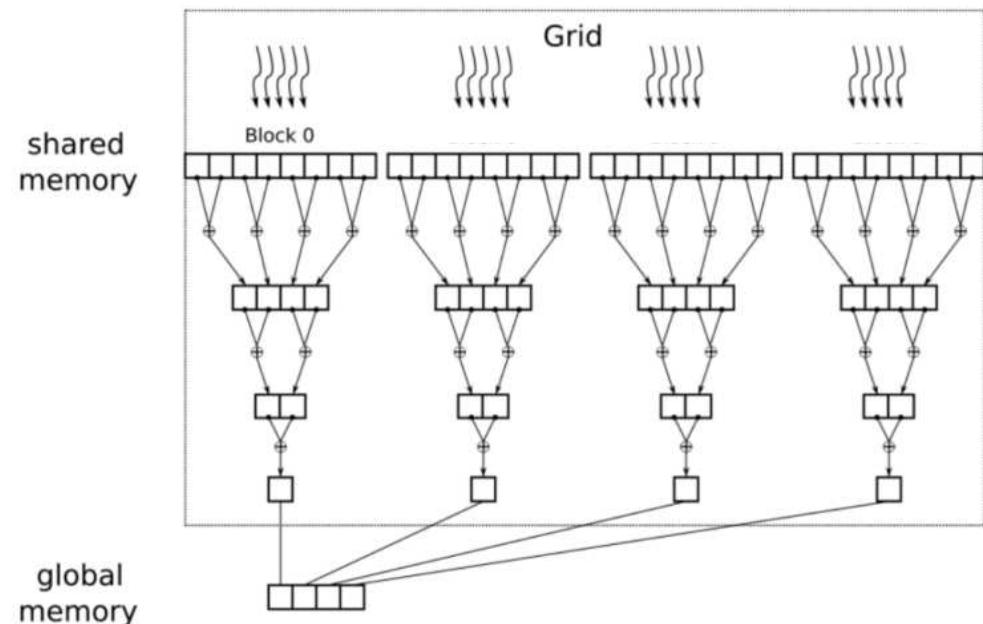
- Fast parallel aggregation step
- Utilizes shared, global and constant memory
- Coalesced memory access
- Almost no thread divergence

Multi-GPU solution

Performance optimized

*bulk aggregations*

*2-step prefiltering*

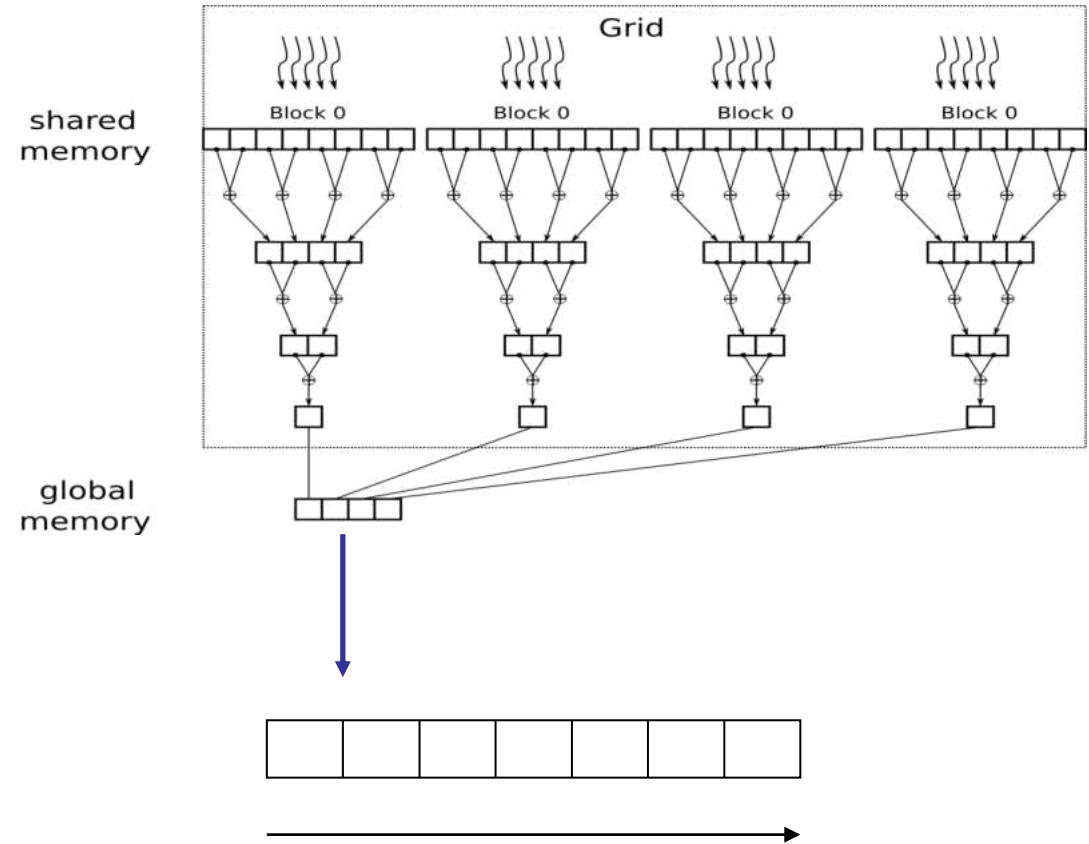


# Target-Driven Aggregation

## (3) Aggregation on GPU

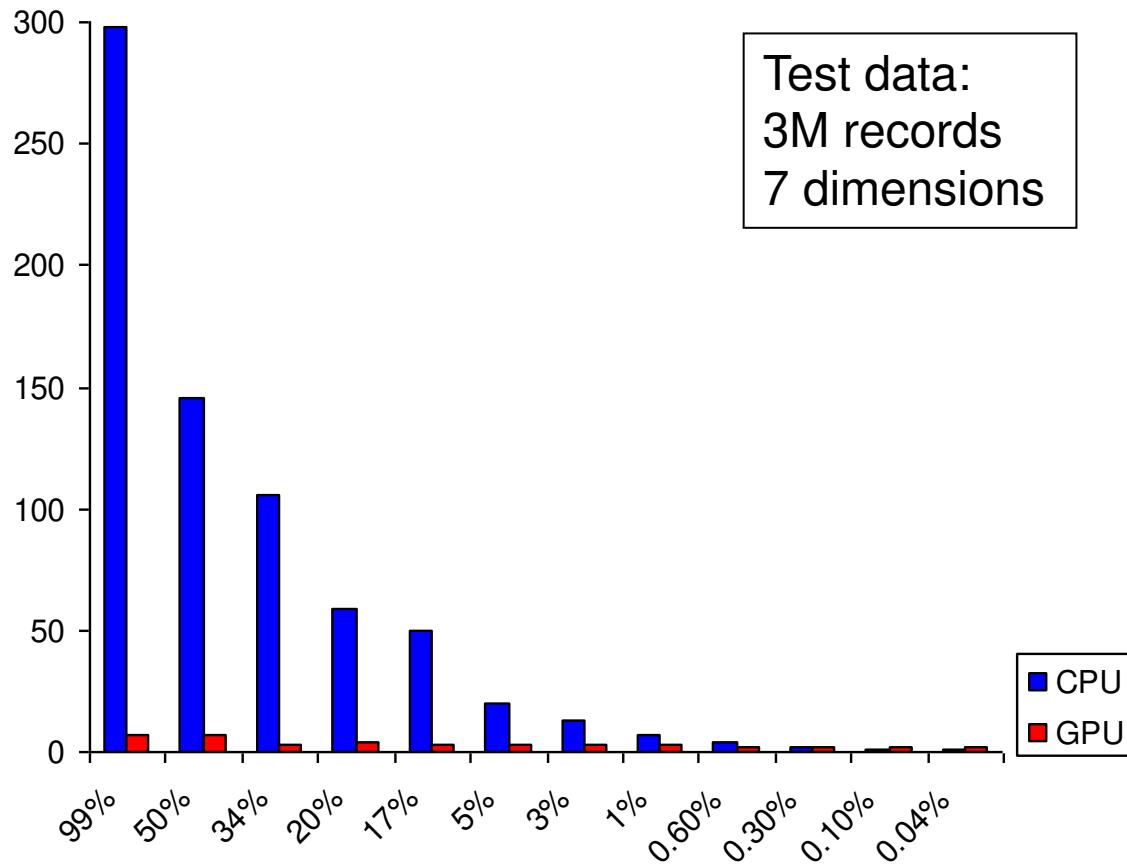
- *Parallel reduction*
- *Completely done in shared memory of GPU*

***Copy result back to CPU***



## (4) Final summarization on CPU

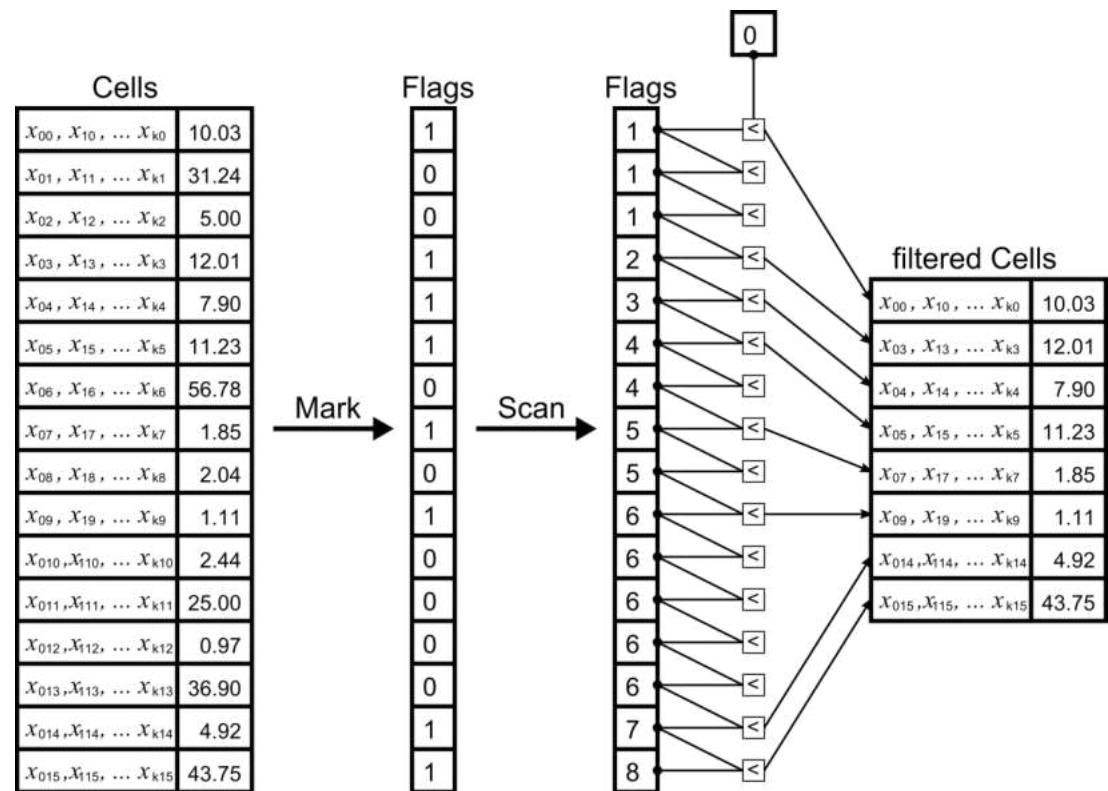
# Timing results



Selectivity	CPU	GPU	Speedup
99%	298ms	7ms	42.6x
50%	146ms	7ms	20.6x
34%	106ms	3ms	35.3x
20%	59ms	4ms	14.8x
17%	50ms	3ms	16.6x
5%	20ms	3ms	6.7x
3%	13ms	3ms	4.3x
1%	7ms	3ms	2.3x
0.6%	4ms	2ms	2.0x
0.3%	2ms	2ms	1.0x
0.1%	1ms	2ms	0.5x
0.04%	1ms	2ms	0.5x

# Optimizing groups of queries

- Fact table prefiltering
  - “*Stream compaction*”
- Simultaneous calculation of multiple queries
  - *Fewer kernel launches*



# Sparse target areas

---

- Sparsity: most cell values in an area of computed cells are 0, because no underlying cells exist
- With target-driven aggregation, those are still „computed“
  - *Lookup of base cells*
  - *Non-negligible cost*
- Too expensive: 0-value computation dominates processing time in sparse areas

# Handling large sparse areas

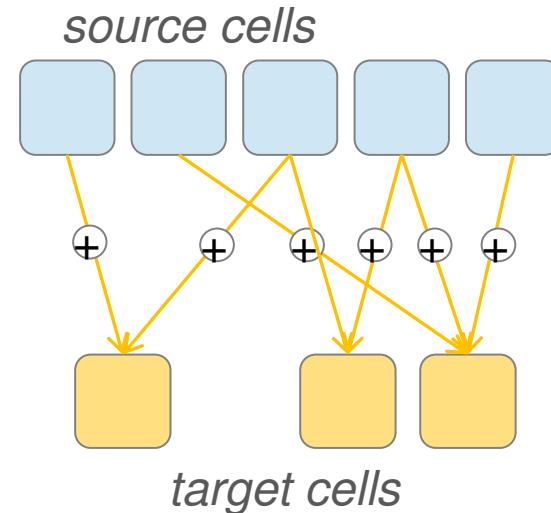
- Requirements

*Performance and memory consumption that correlate with number of **non-zero** target cells*

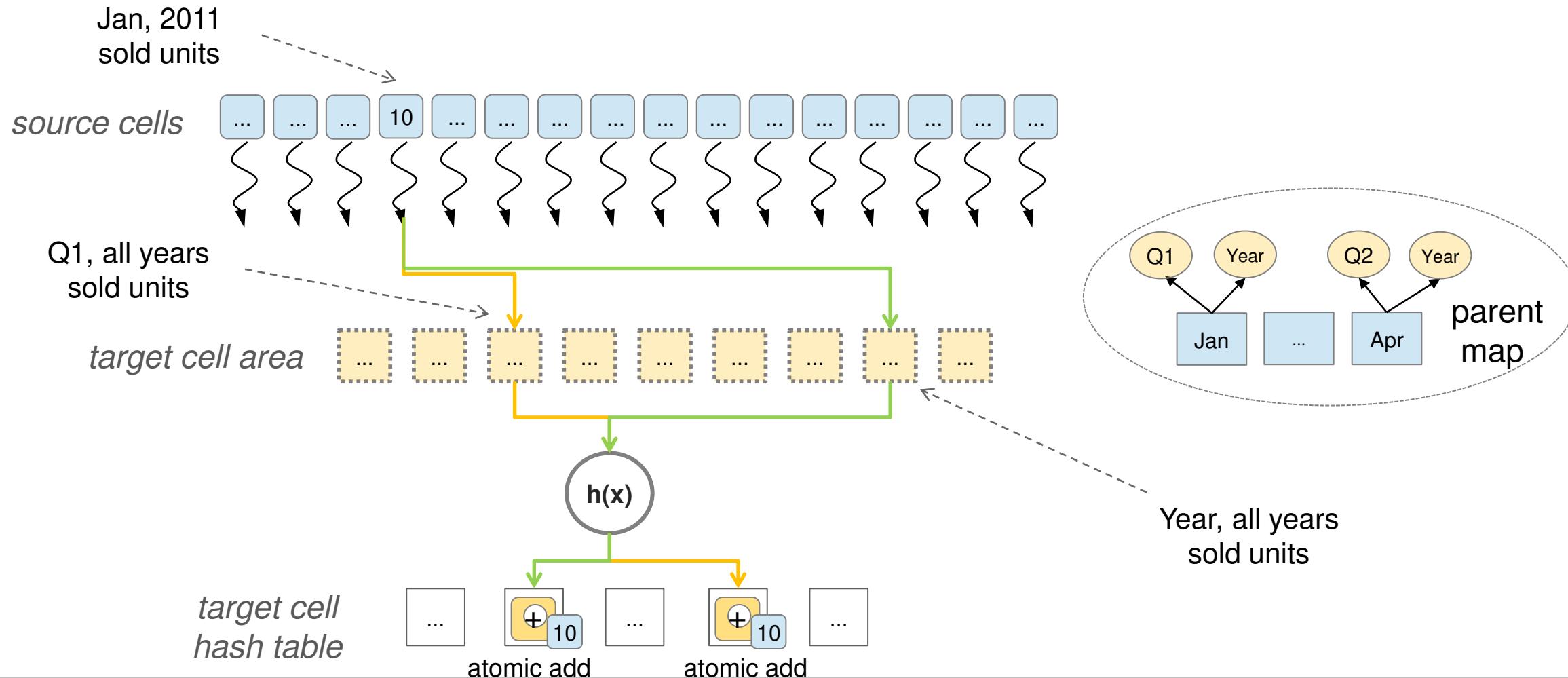
- Solution

*Source-driven approach*

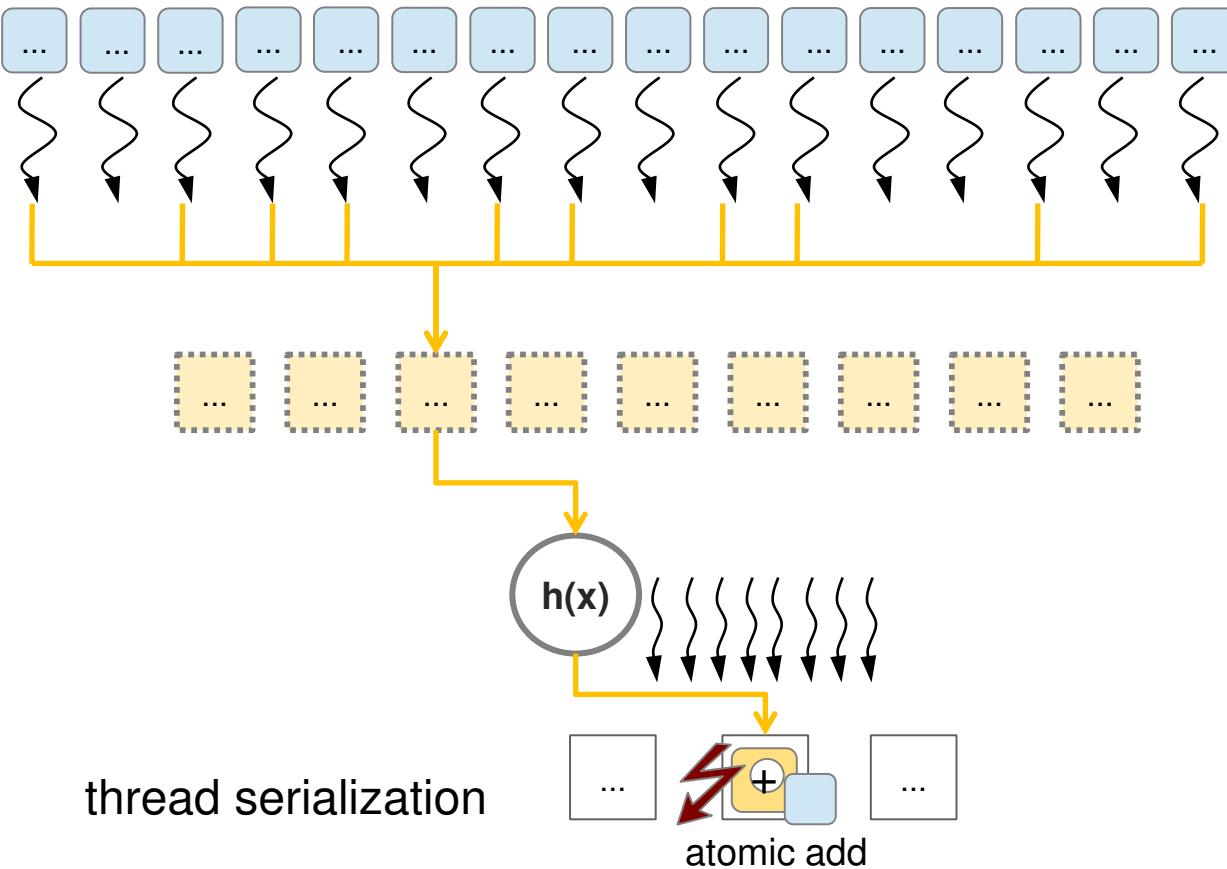
- *Serialized aggregation with atomics*
- *Utilize hash tables on GPU*



# Source-driven aggregation

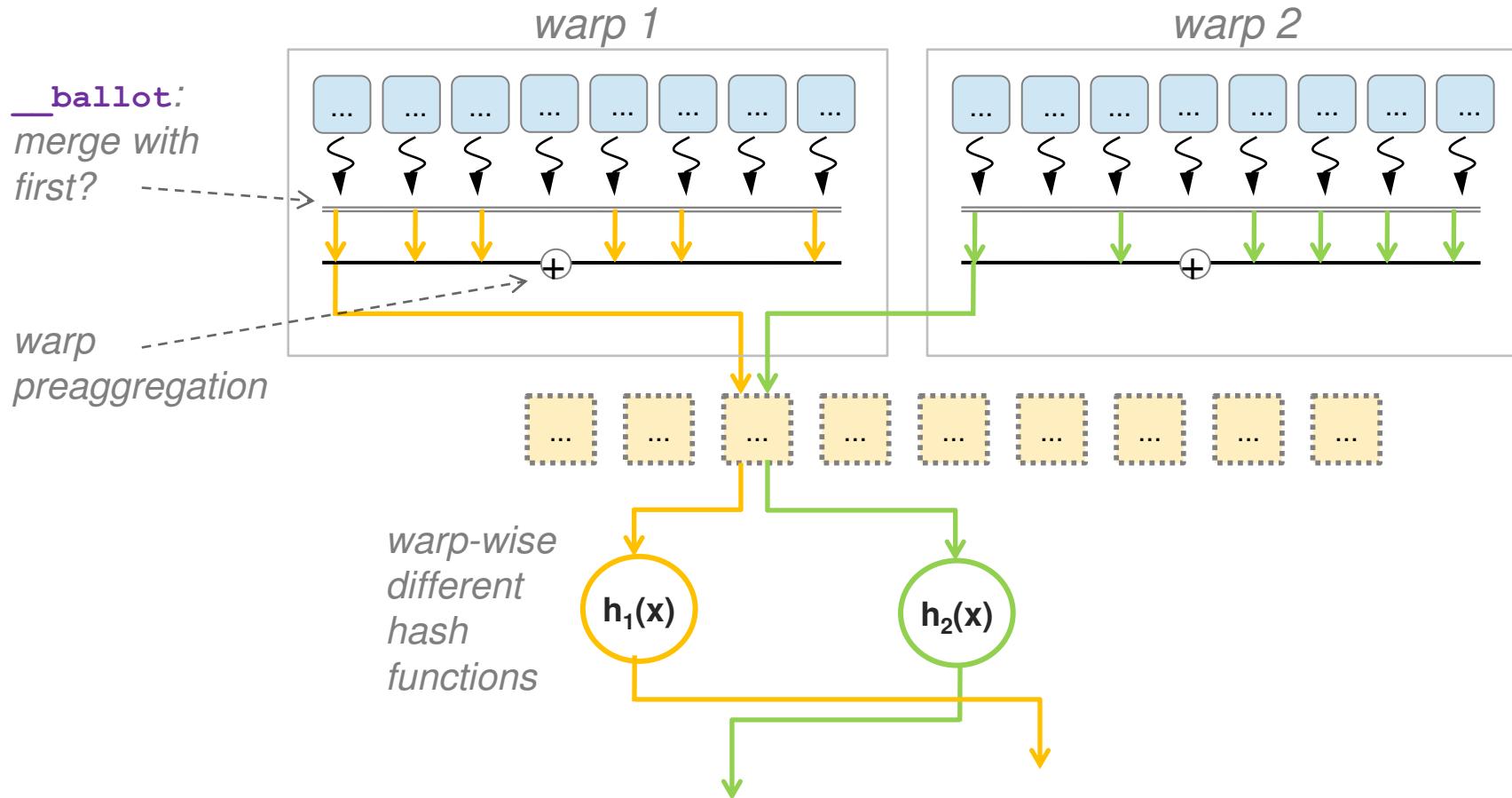


# Atomics: Contention



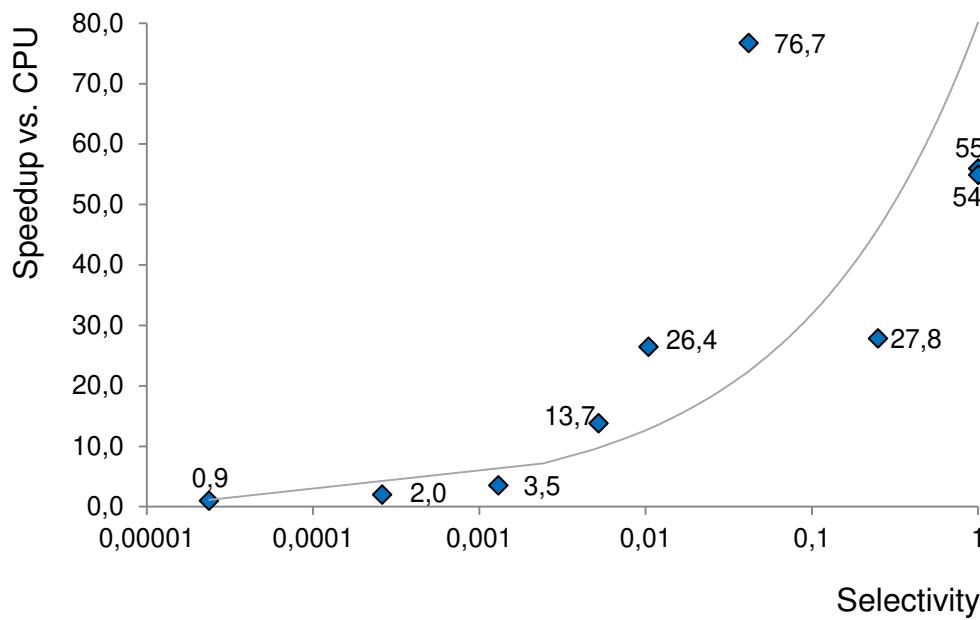
*Great improvement  
in Fermi and Kepler  
over CC 1.X*

# Reducing contention

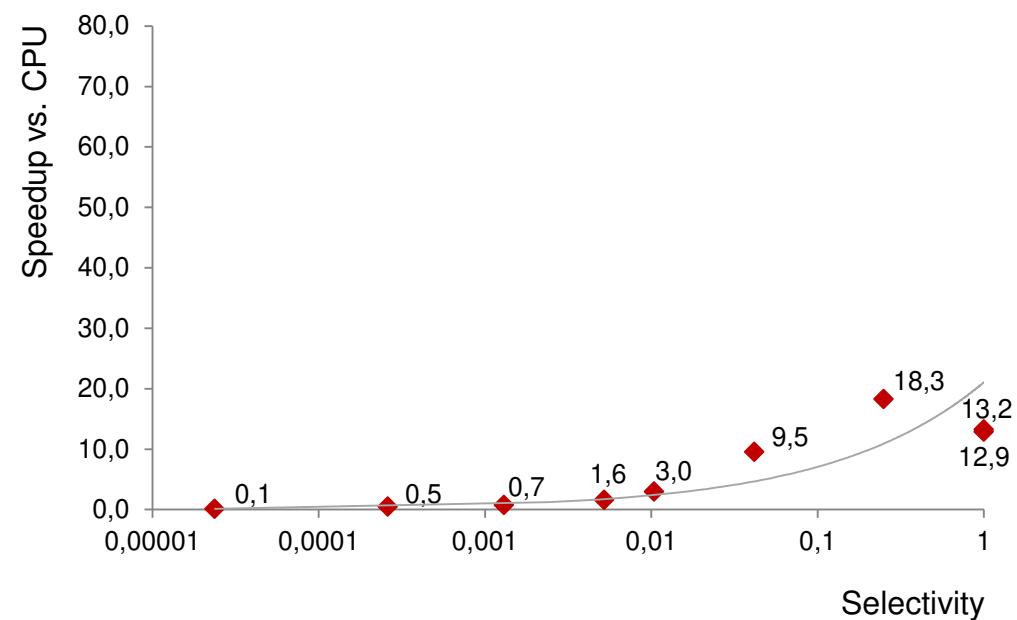


# Speedup: Small target areas (1-11 cells)

GPU Target Driven



GPU Source Driven

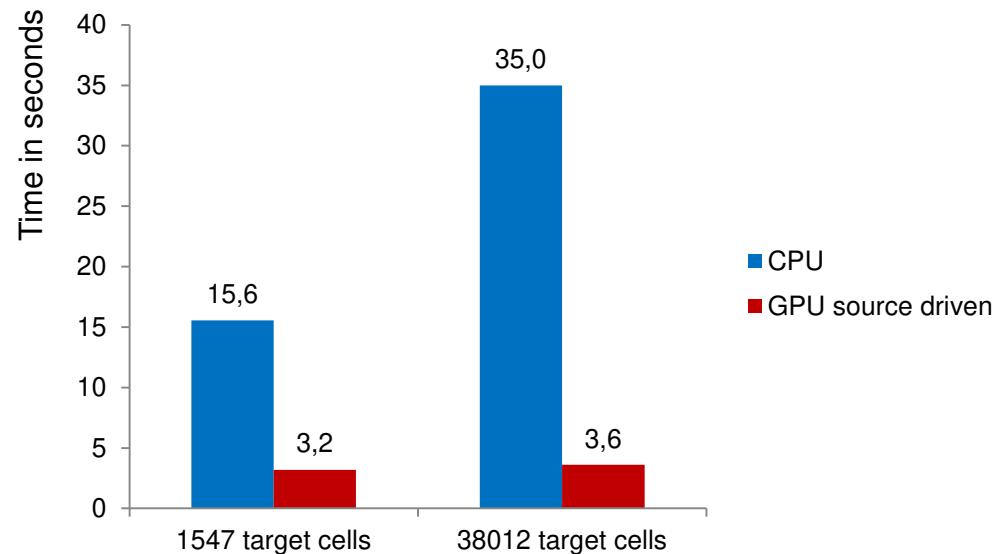


Database: 1B records (filled cube cells)

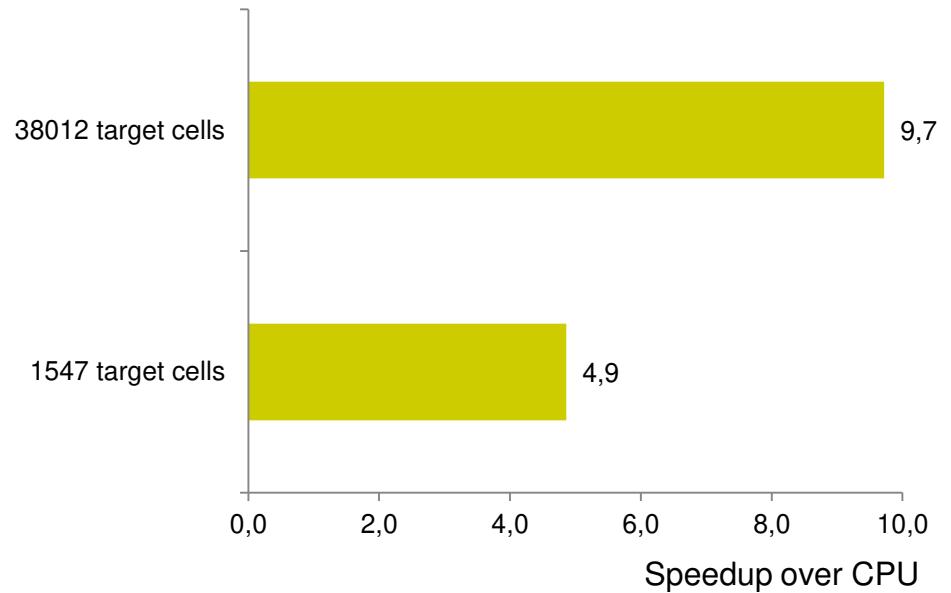
GPU: 3x Tesla C2070 (18 GB RAM)

# Larger areas

Calculation times



Speedup

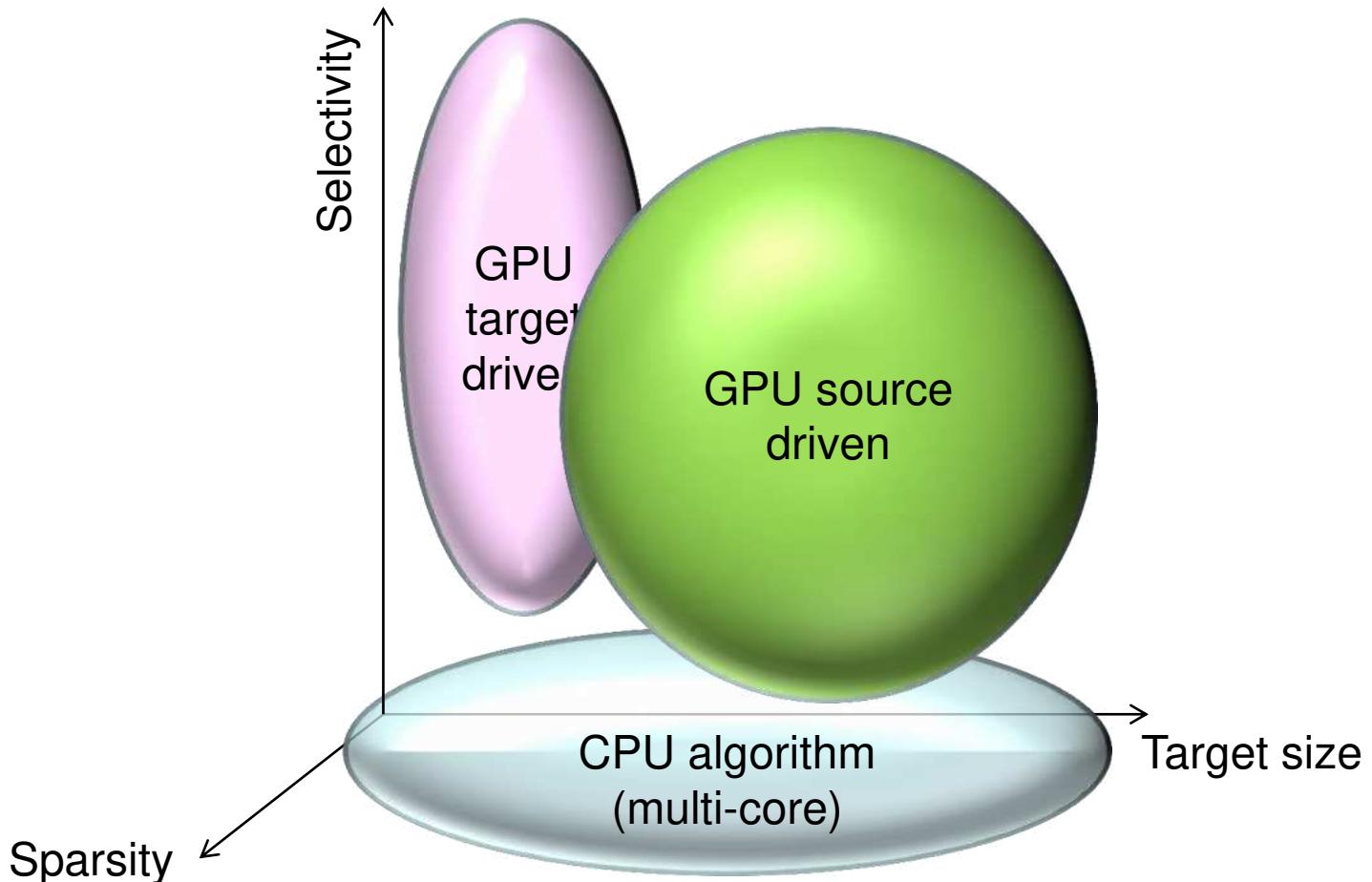


Database: 40M records (filled cube cells)

GPU: 2x Tesla K20 (10 GB RAM)

# Comparison: aggregation algorithms

- CPU algorithm good for low aggregation
- Target driven algorithm: good for small and/or dense target areas
- Source driven algorithm: optimized for large and sparse areas



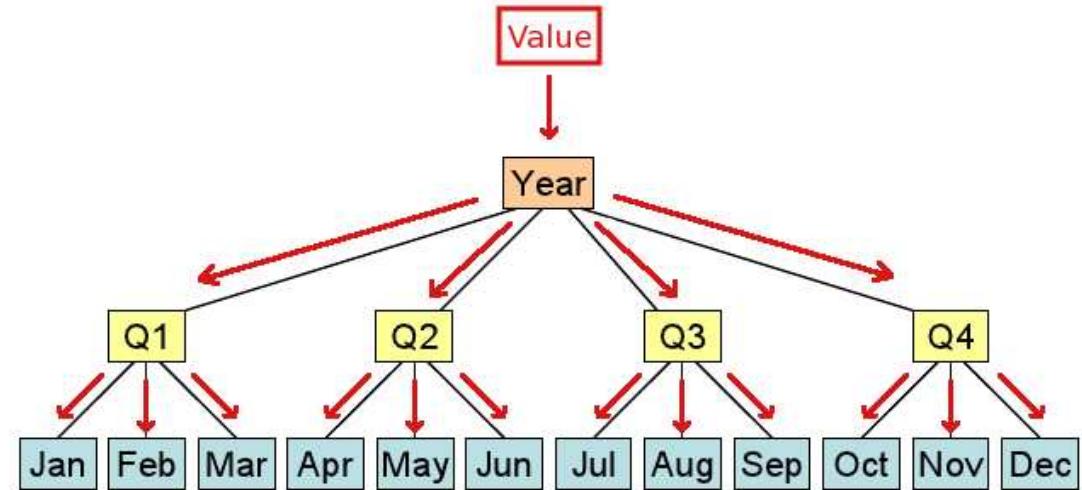
# Remaining challenges

---

- Decision mechanism
  - *When GPU, when CPU?*
  - *Which GPU algorithm in which situation?*
- Find suitable thresholds
- What about large and *dense* target areas?  
→ GPU memory problem

# Writeback in Top-Down Planning

- Writeback: "opposite direction" of aggregation (disaggregation)
- Value inserted at high level of aggregation is broken down to lower levels until the base level
- All underlying base cells are modified, depending on the type of writeback



# Ranges and areas

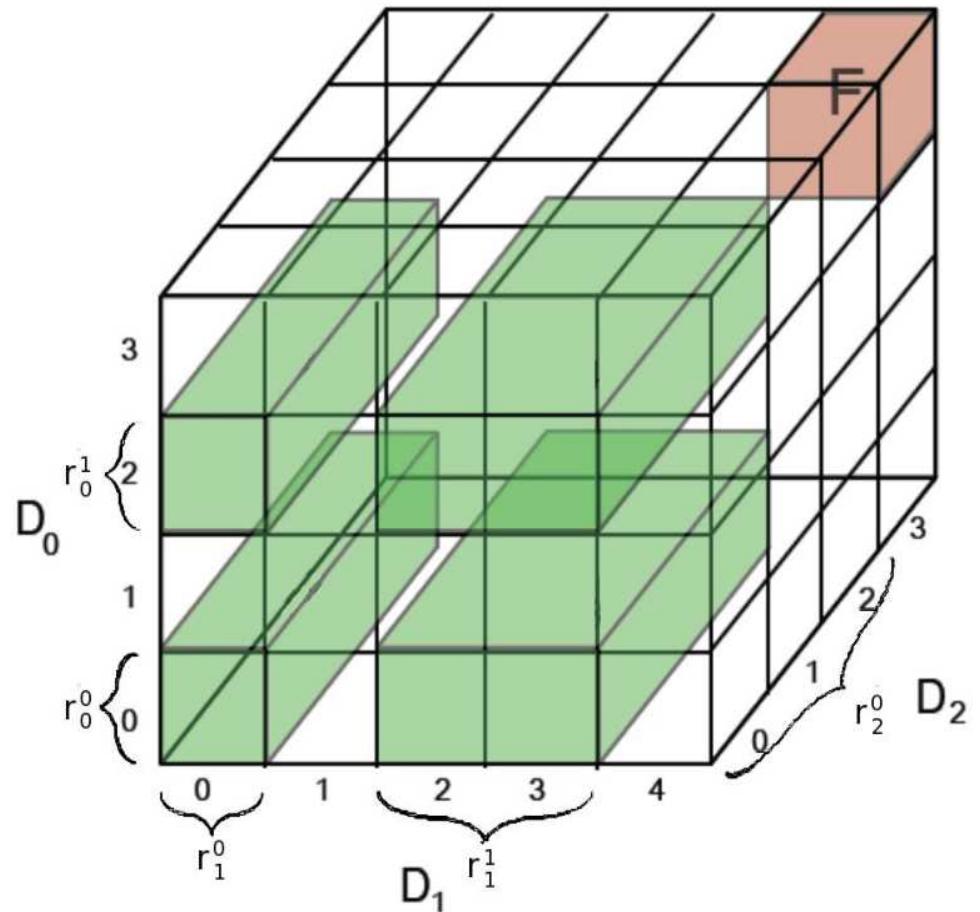
- Base elements in each dimension are collected in **ranges**

$$D_0: \{ [0,0], [2,2] \} \quad |D_0| = 2$$

$$D_1: \{ [0,0], [2,3] \} \quad |D_1| = 3$$

$$D_2: \{ [0,2] \} \quad |D_2| = 3$$

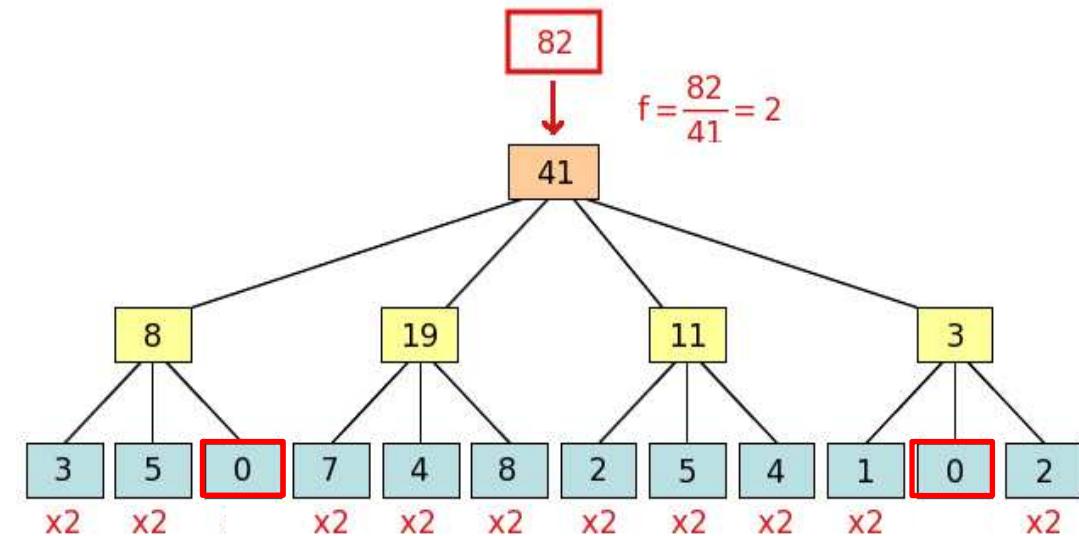
- The Cartesian product of ranges across all dimensions forms an **area**  
 $D_0 \times D_1 \times D_2$



# Multiply-base distribution

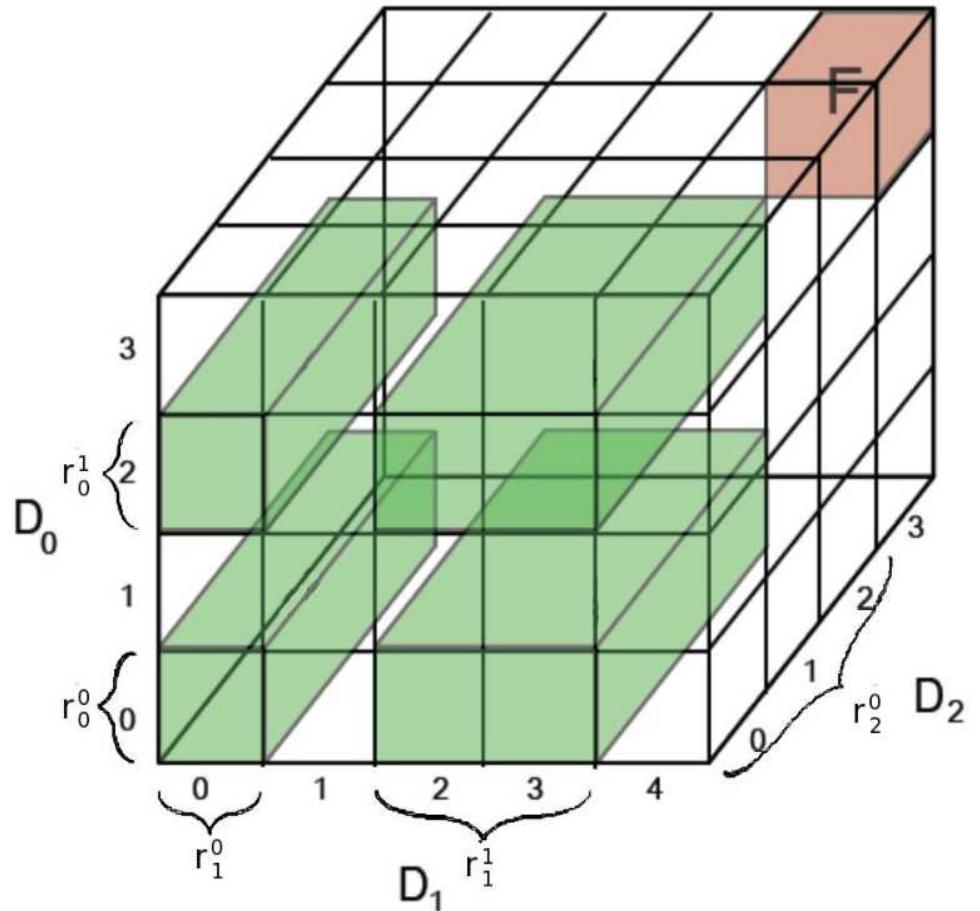
- An existing value  $v_{old} \neq 0$  on an aggregated level is set to  $v_{new}$ . Every underlying base cell is multiplied by

$$f = \frac{v_{new}}{v_{old}}$$

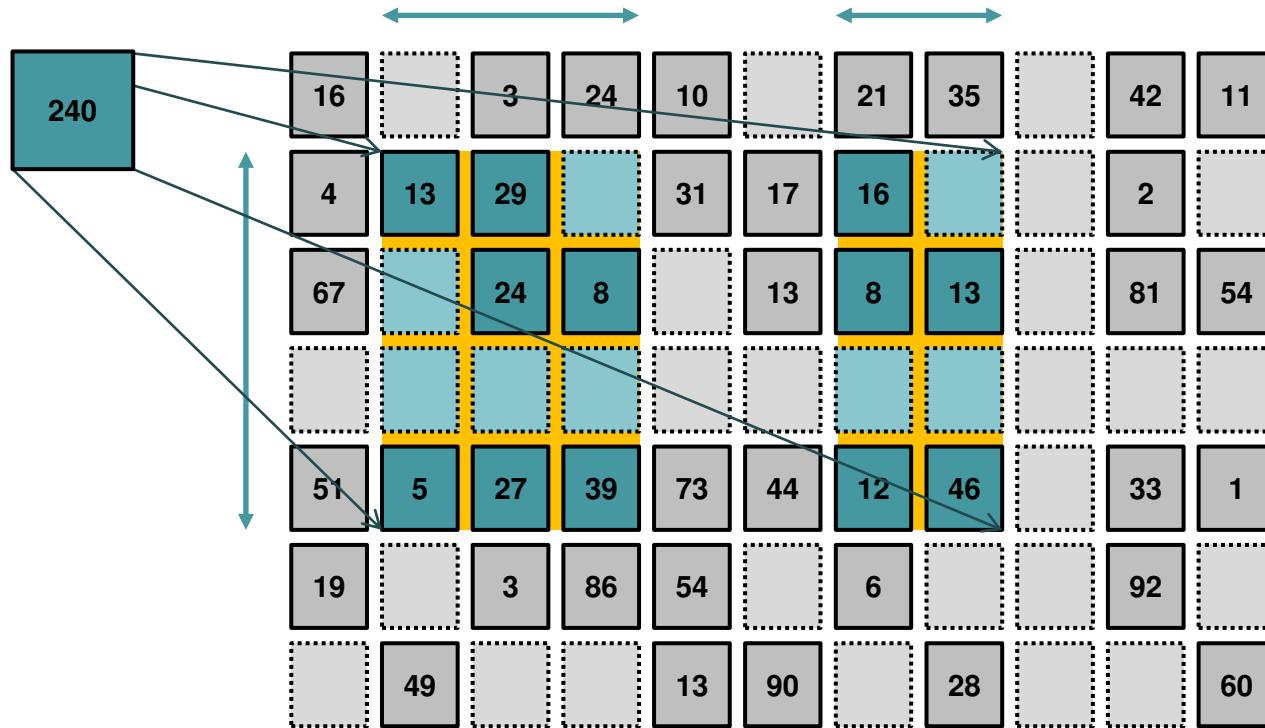


# Multiply-base distribution

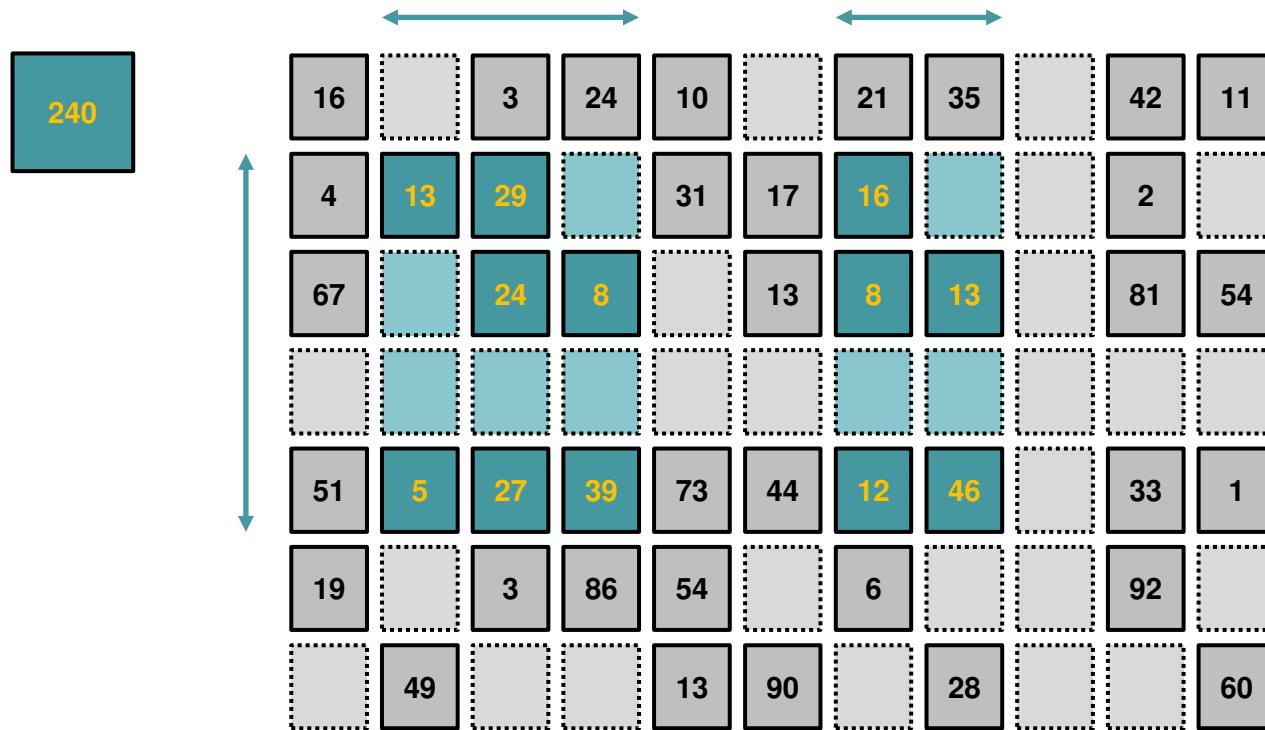
	$D_0$	$D_1$	$D_2$	
	$D_0$	$D_1$	$D_2$	
Thread 0	0	0	0	6
Thread 1	0	0	1	10
Thread 2	0	0	2	14
Thread 3	0	1	0	4
Thread 4	0	1	1	8
Thread 5	0	1	2	2
Thread 6	0	2	0	10
Thread 7	0	2	1	8
Thread 8	0	2	2	2
Thread 9	0	3	0	4
Thread 10	0	3	1	4
Thread 11	0	3	2	18
Thread 12	0	4	0	3
:	:	:	:	
:	:	:	:	
:	:	:	:	



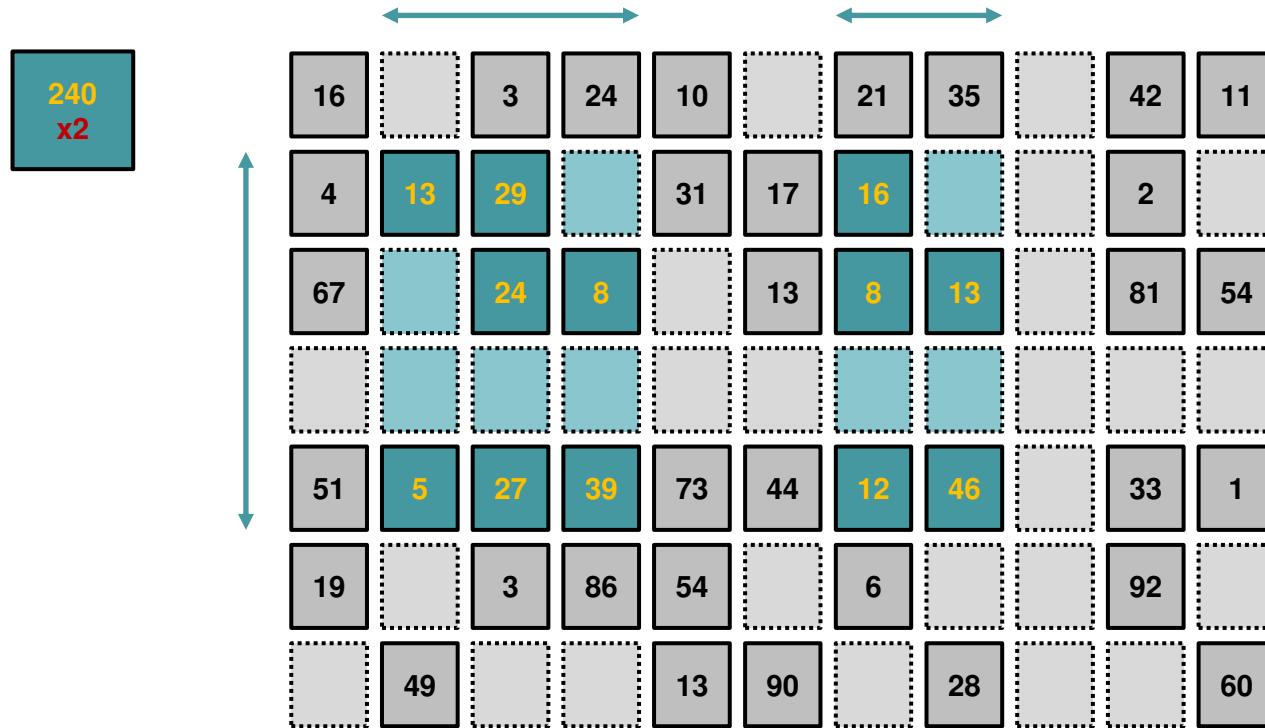
# Multiply-base distribution



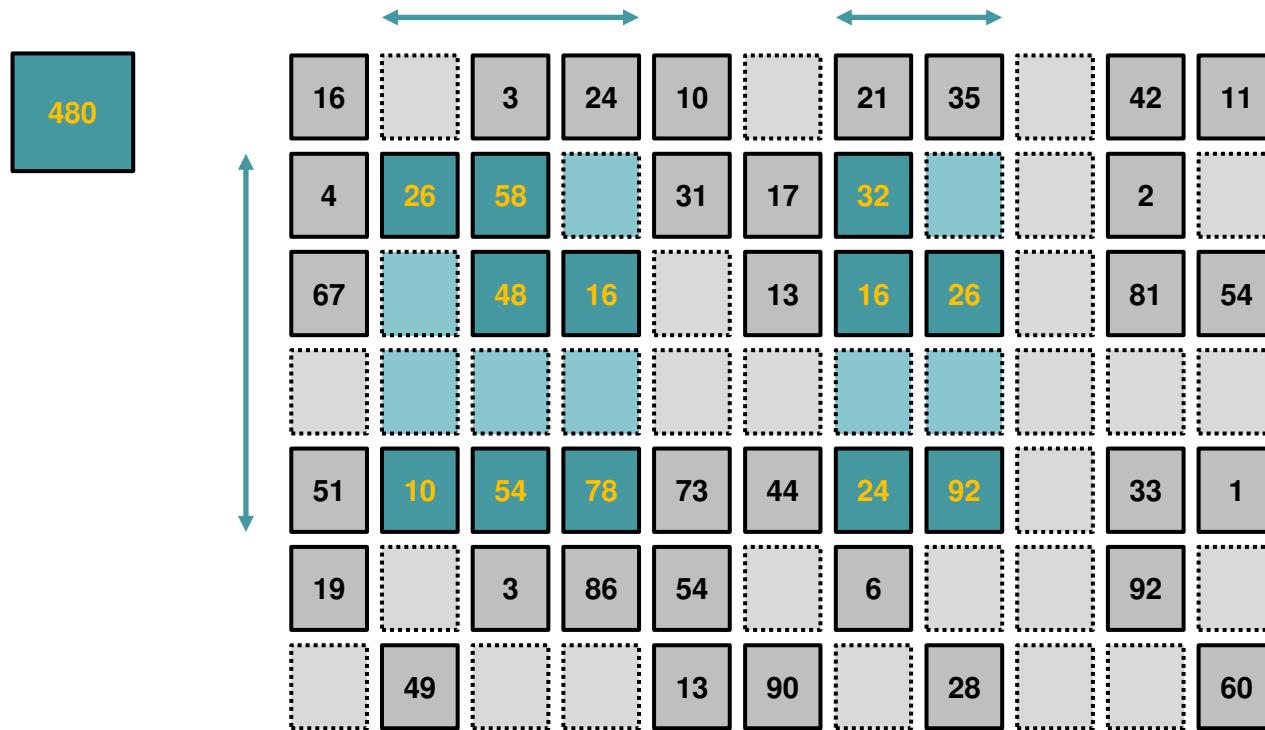
# Multiply-base distribution



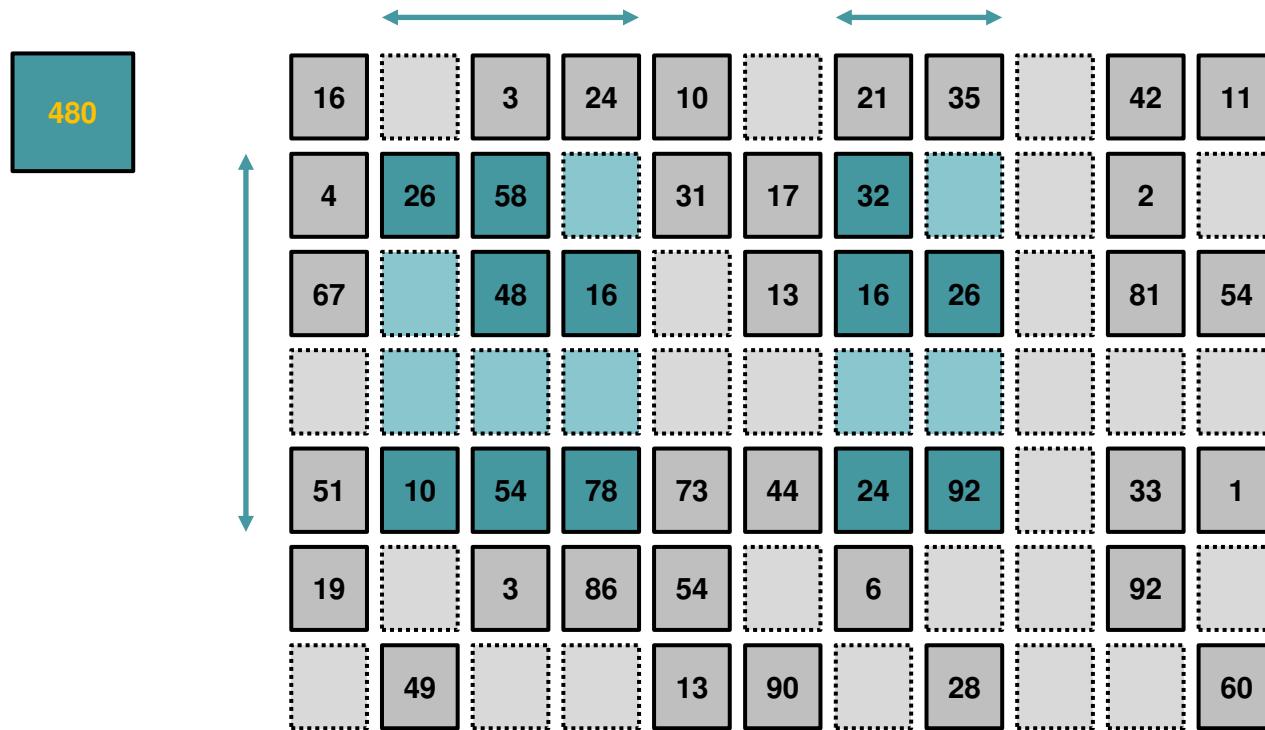
# Multiply-base distribution



# Multiply-base distribution

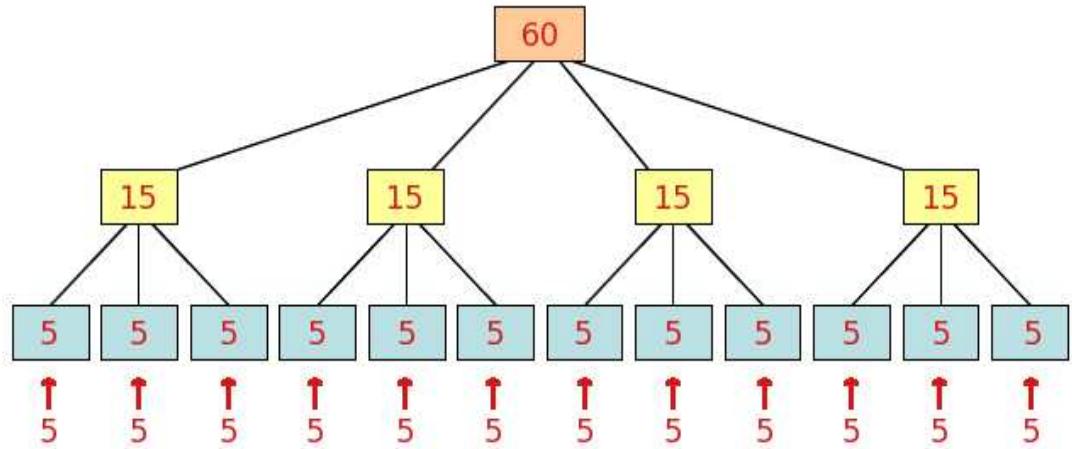


# Multiply-base distribution

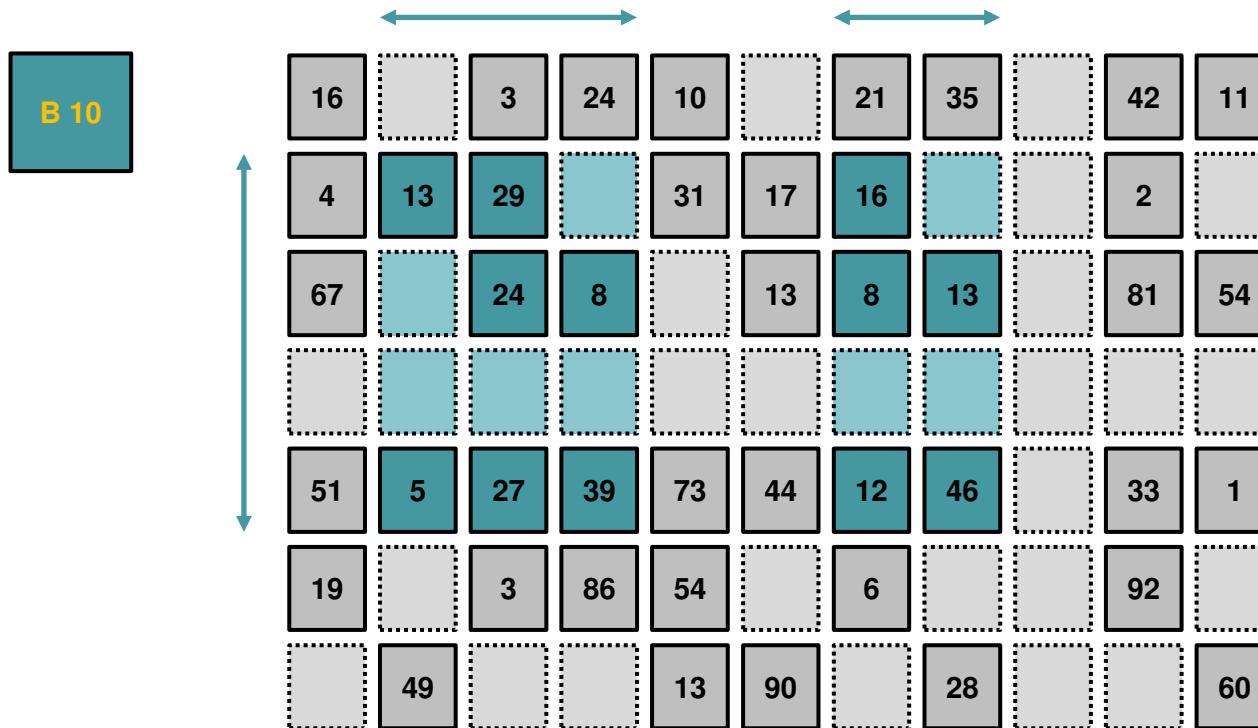


# Set-base distribution

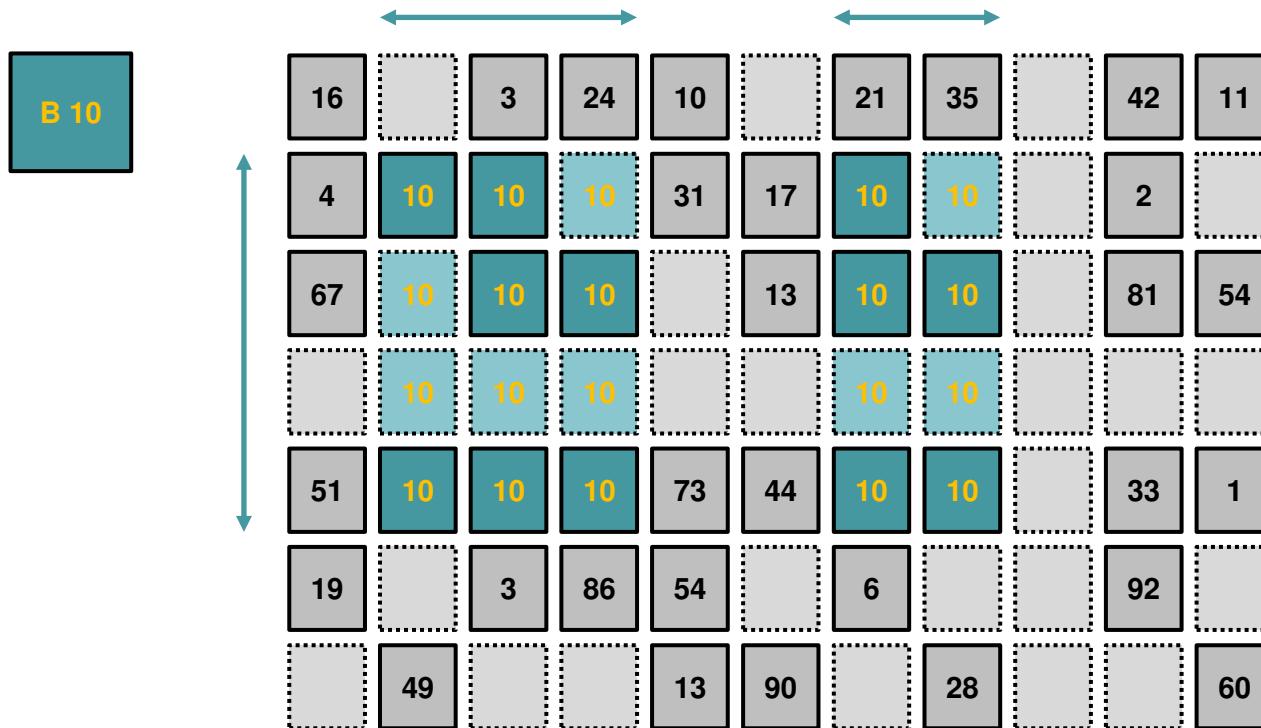
- Every relevant base cell in the area is set to the same **given value**
- Naïve approach:  
search for all relevant paths and replace cell values
  - *Problem: what about zero-value cells, which are not represented?*
- Better approach:
  - (1) *Delete all existing cells in area*
  - (2) *Create all cells in area with new value*



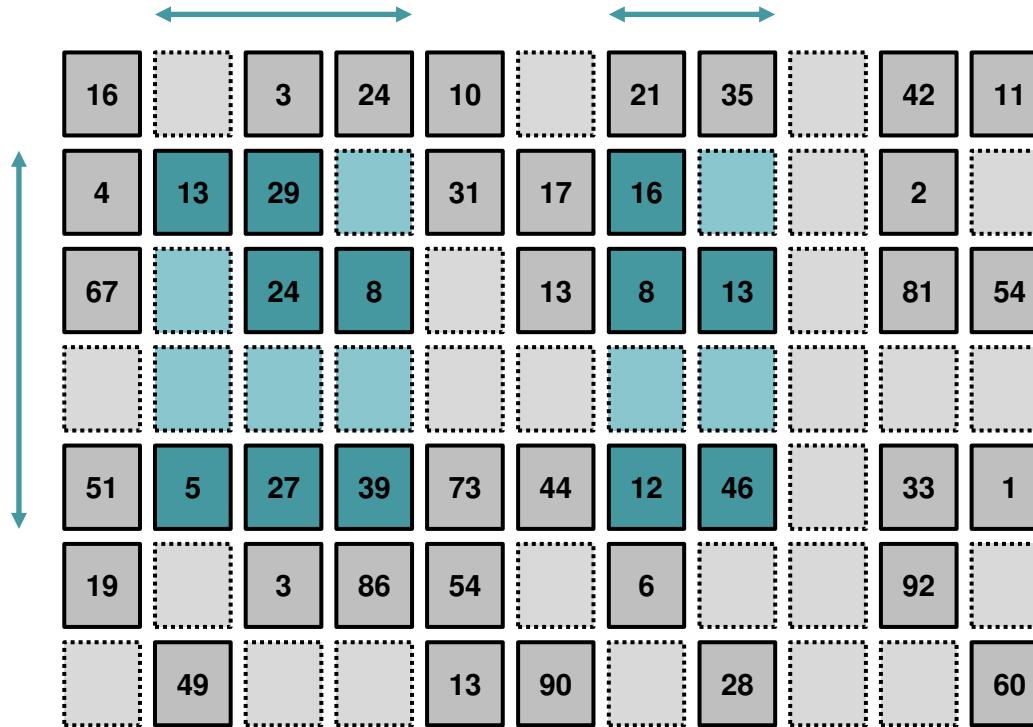
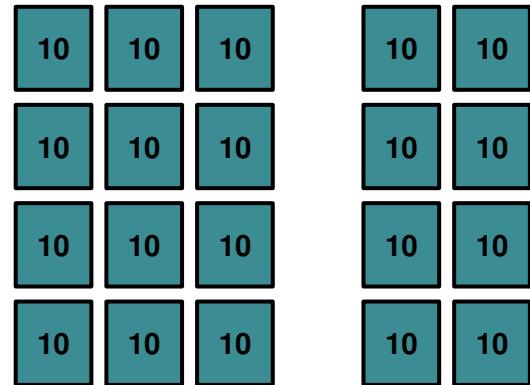
# Set-base distribution



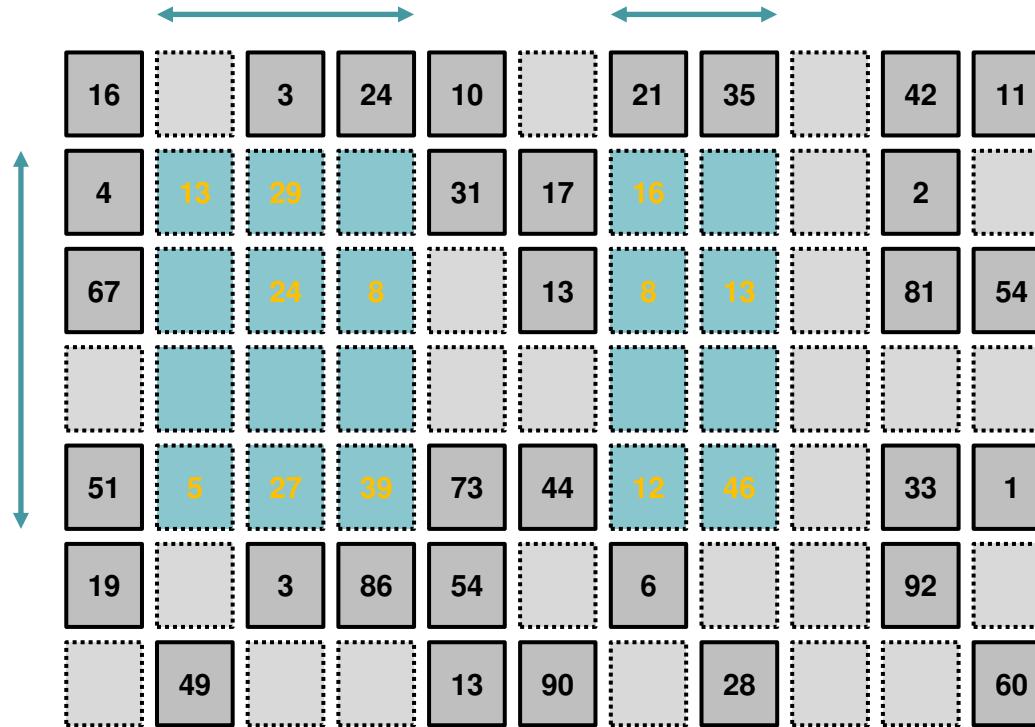
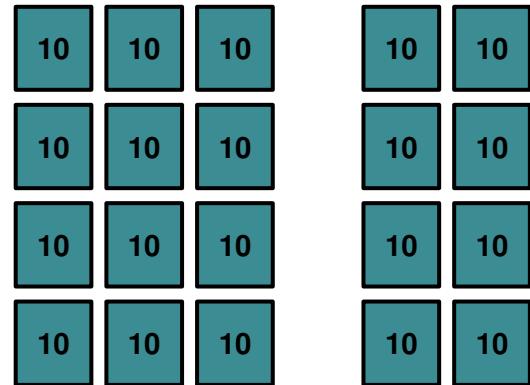
# Set-base distribution



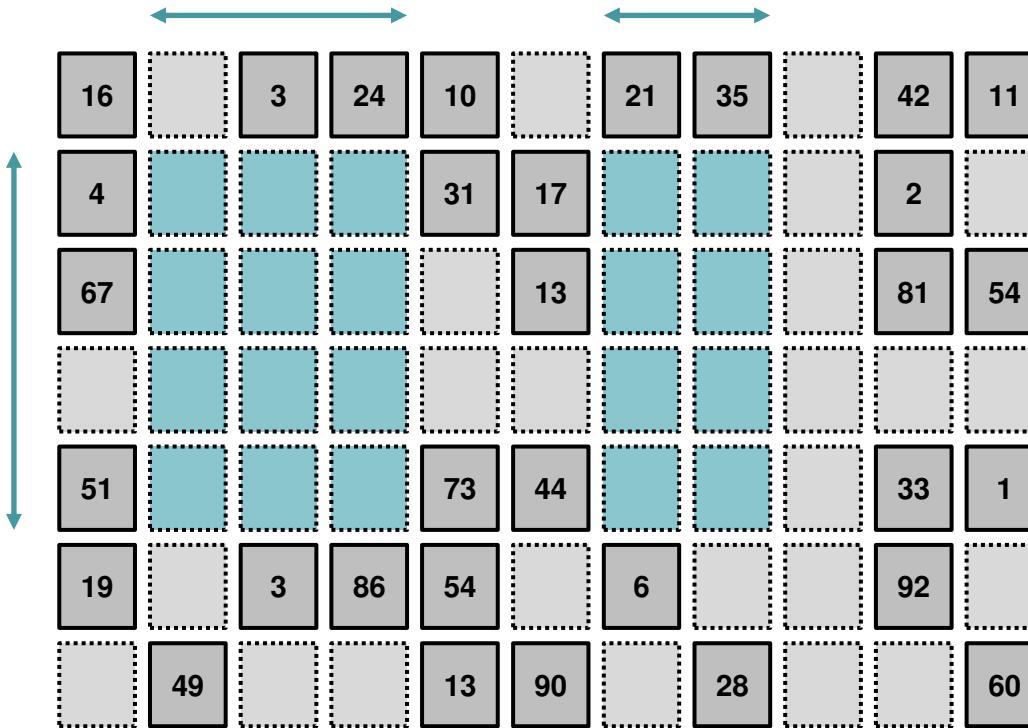
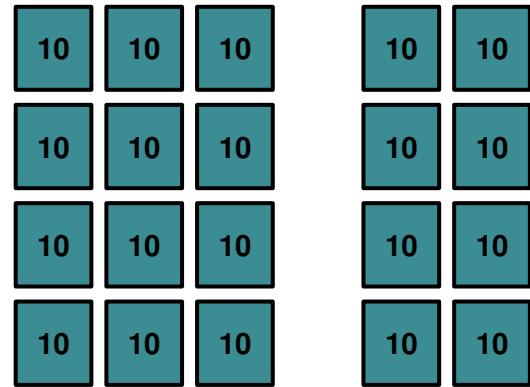
# Set-base distribution



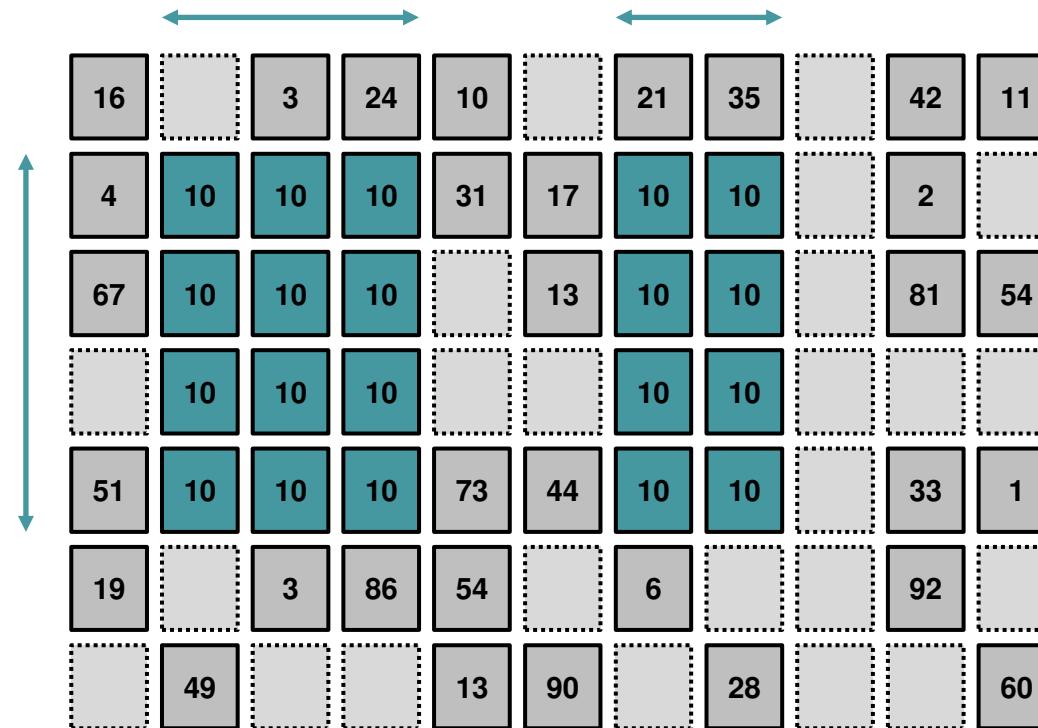
# Set-base distribution



# Set-base distribution

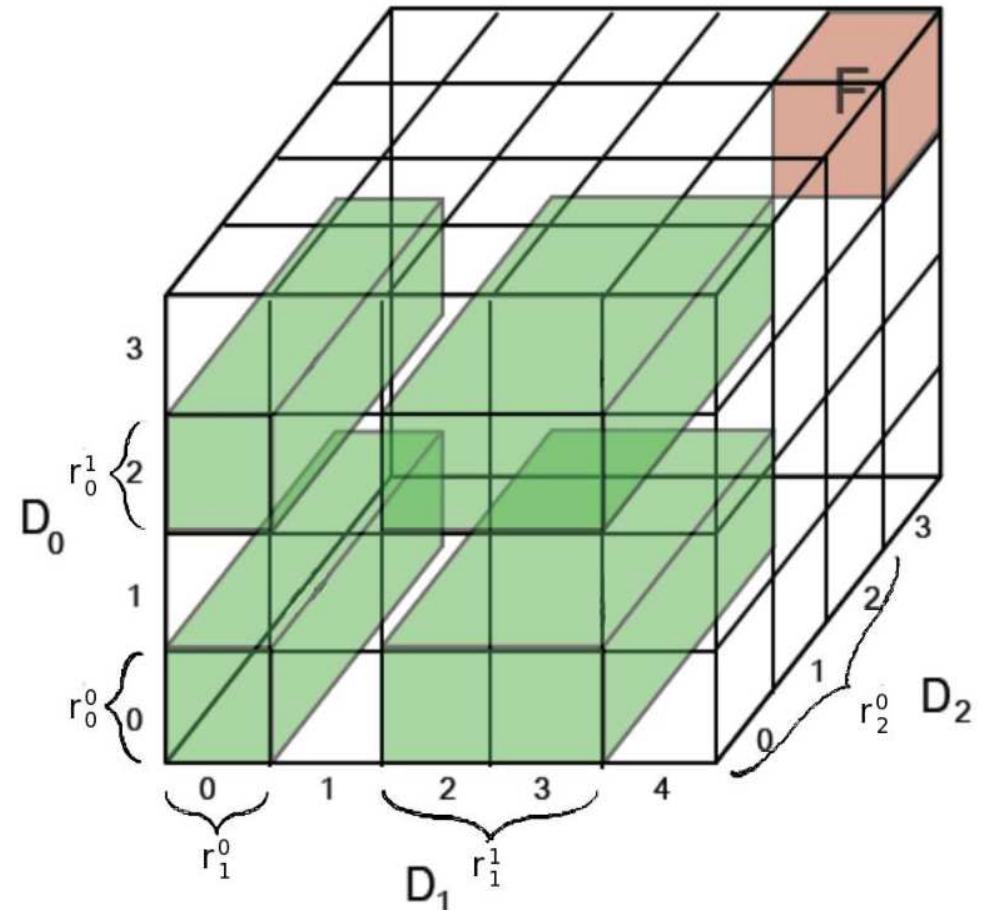


# Set-base distribution

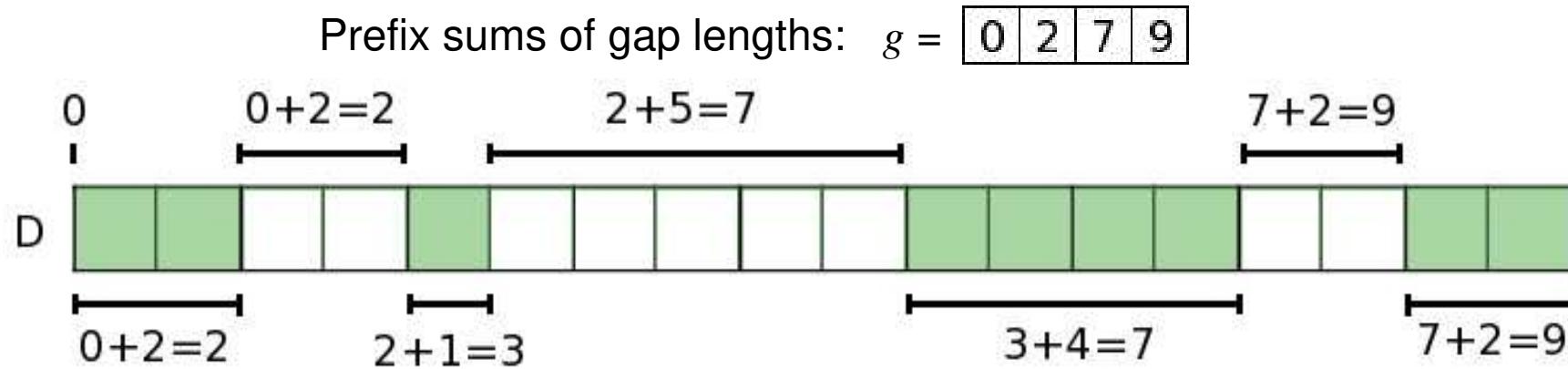


# Parallel creation of all cell paths in an area

- “Parallel enumeration” of the area:
  - *Each thread computes the path of “its” cell from the thread ID*
  - **Problem:**
    - Gaps between ranges of a dimension prevent simple iterations
    - Iterating over all ranges and counting all visited elements is inefficient
  - *Solution:*
    - Represent ranges by pre-calculated **prefix sums** (rather than start and end points)



# Prefix sum representation of ranges

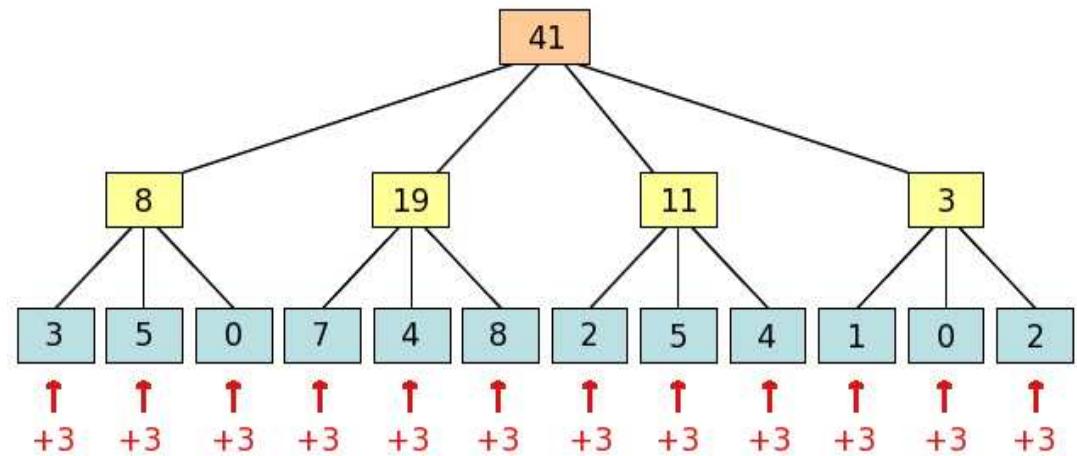


Prefix sums of range lengths:  $r = \boxed{2 \ 3 \ 7 \ 9}$

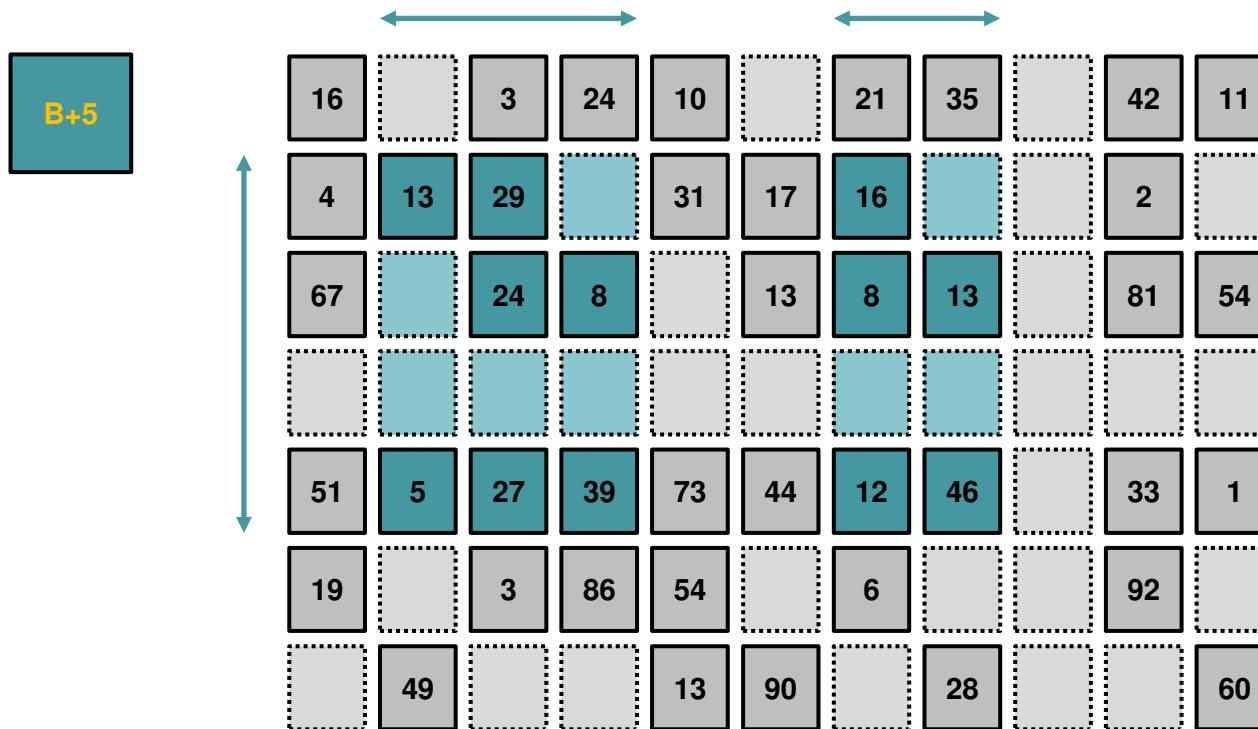
Index  $i$  of  $k^{\text{th}}$  relevant element in  $D$ : (1) Find smallest  $m$  such that  $r[m] \geq k$   
(2)  $i = g[m] + k - 1$

# Add-base distribution

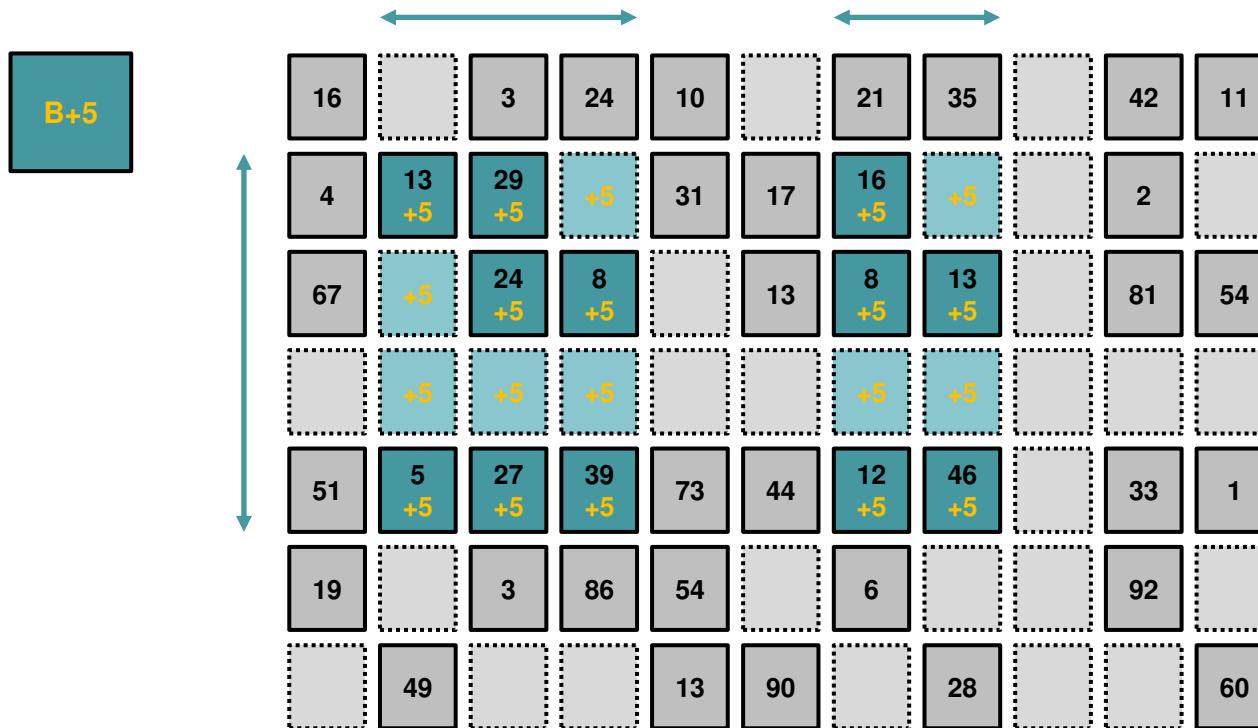
- The same given value  $v$  is **added** to the value of each relevant base cell
- Approach:
  - Create all cells of the area and set value to  $v$  (as before) and store them temporarily
  - Find all previously existing relevant cells and add their (old) value to the one in the new temporary area
  - Delete old relevant cells and persist temporary storage



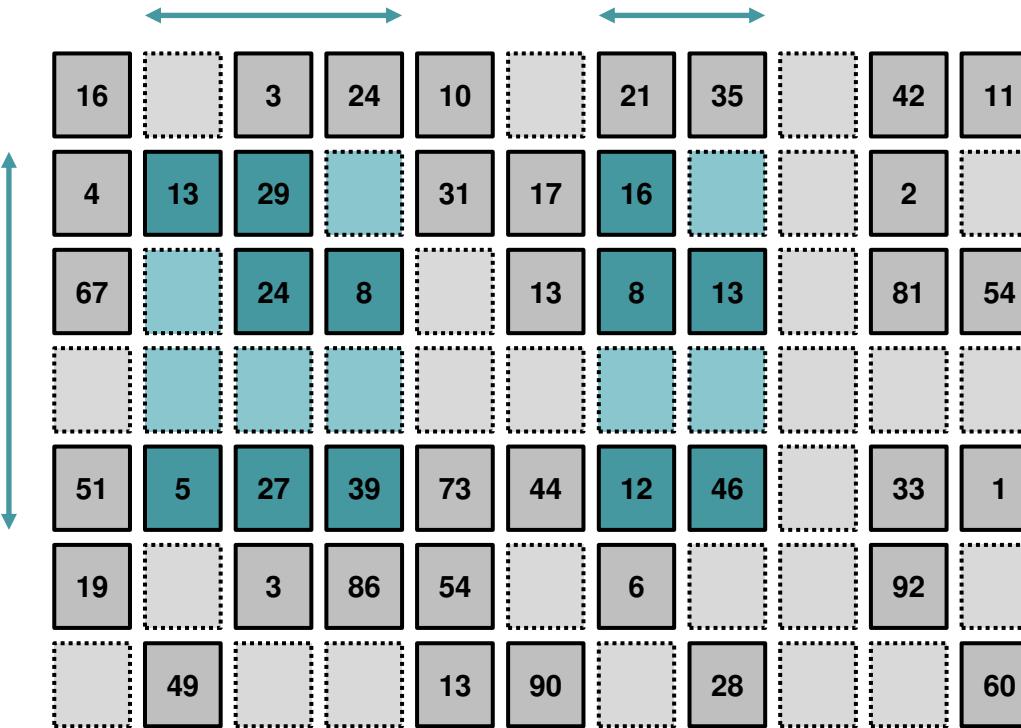
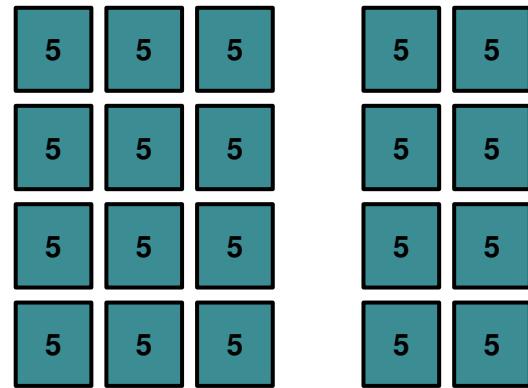
# Add-base distribution



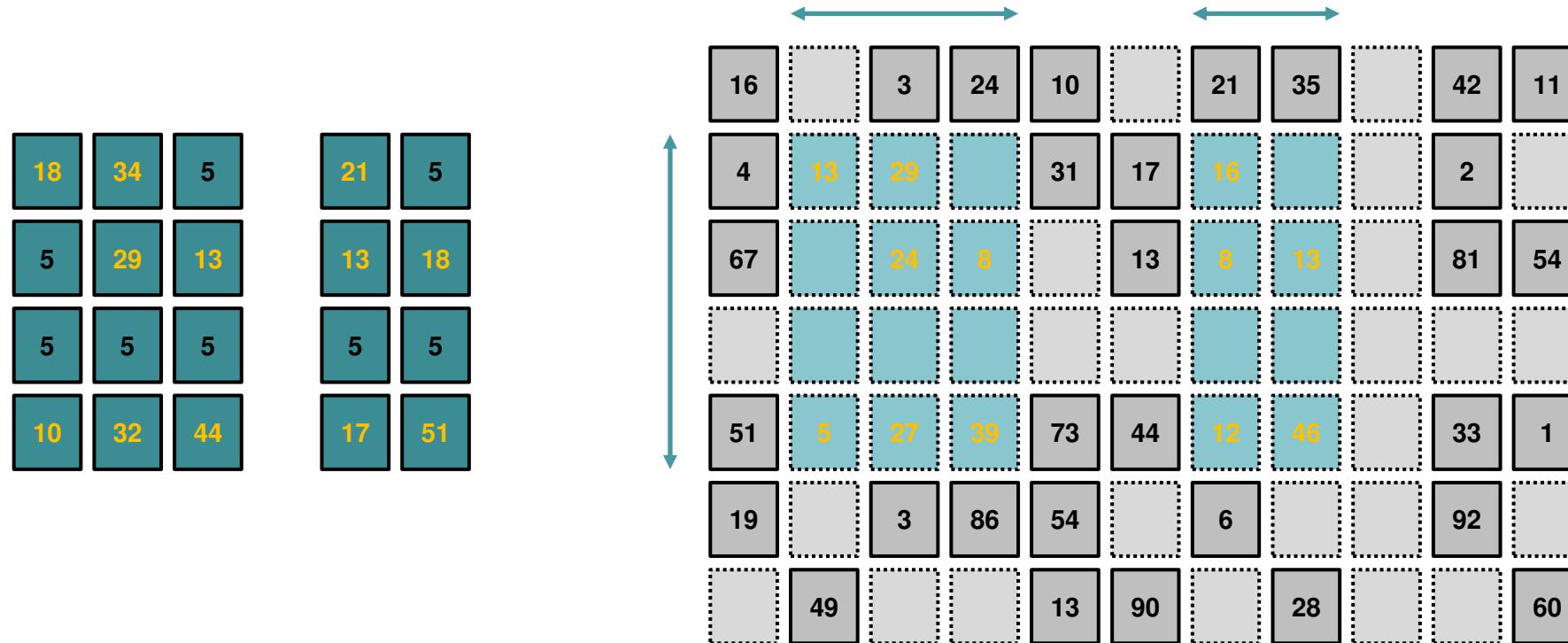
# Add-base distribution



# Add-base distribution



# Add-base distribution

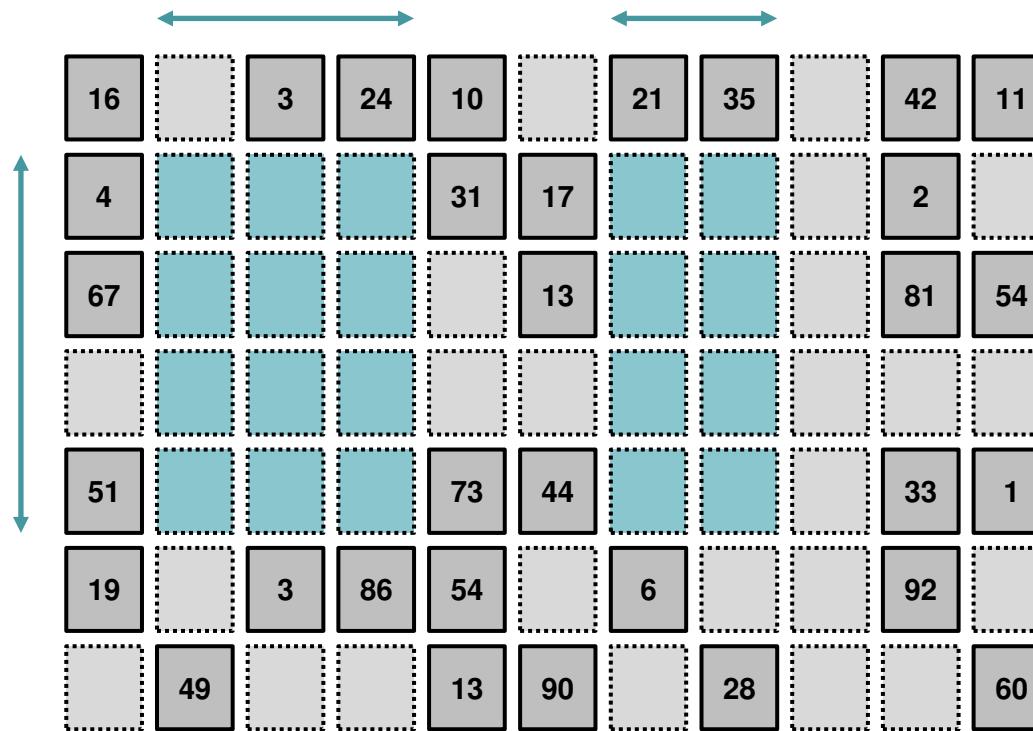


# Add-base distribution

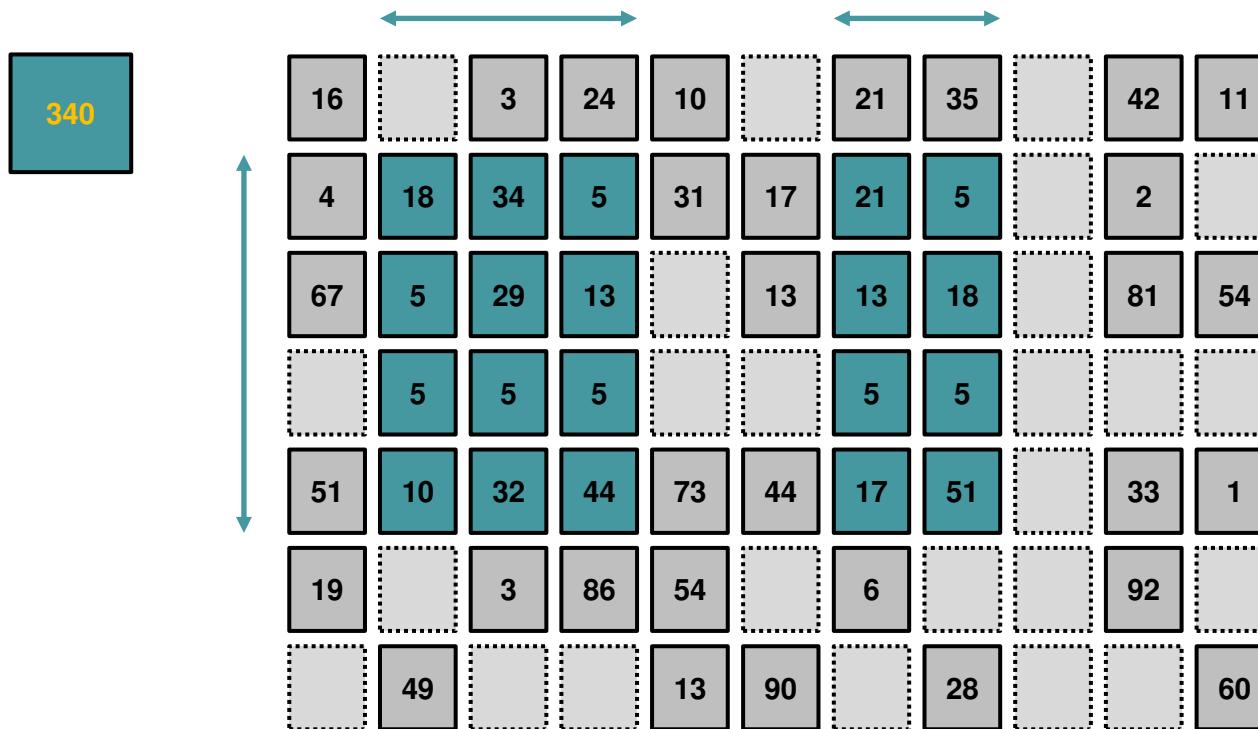
18	34	5
5	29	13
5	5	5
10	32	44

21	5
13	18
5	5
17	51



# Add-base distribution



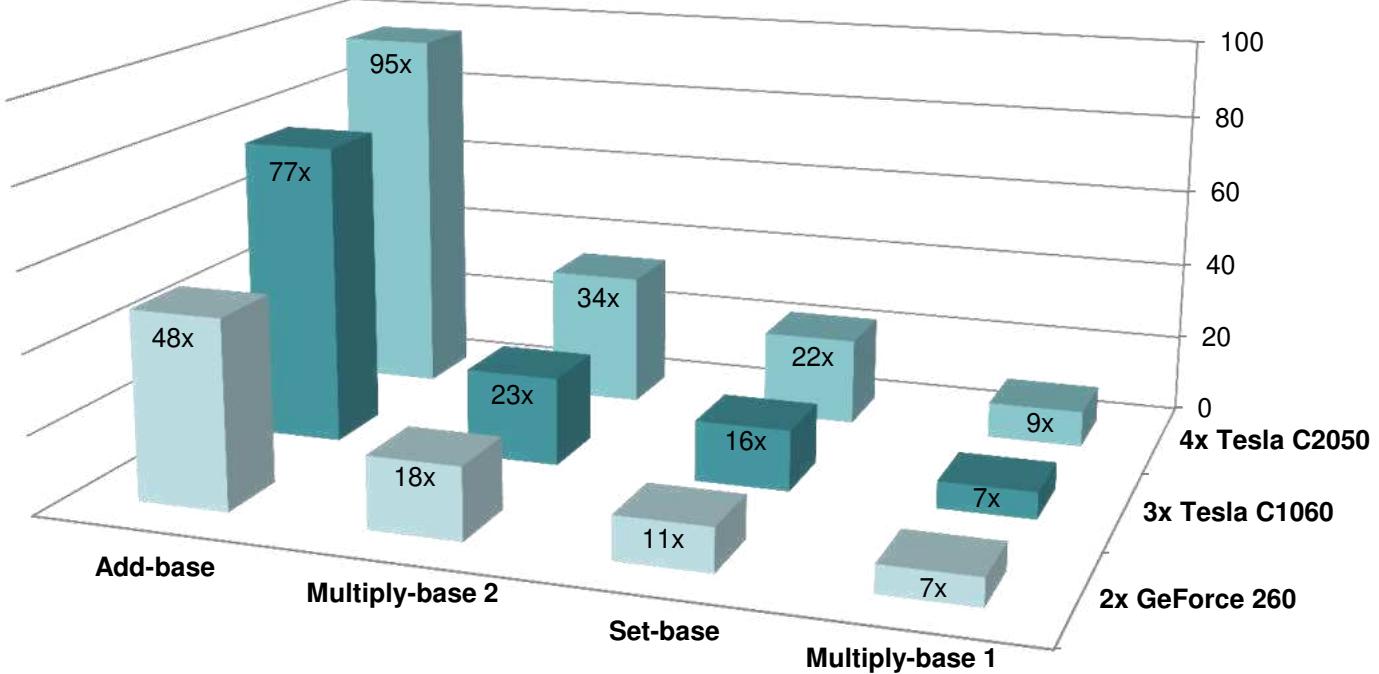
# Performance tests

Timings (in ms)	CPU Intel DualCore	2x GPU GeForce 260	3x GPU Tesla C1060	4x GPU Tesla C2050
Multiply-base 1	3,548	466 (7.6 x)	558 (6.4 x)	435 (8.2 x)
Refresh 1	1,131	200 (5.7 x)	127 (8.9 x)	74 (15.3 x)
<b>Sum</b>	<b>4,679</b>	<b>666 (7.0 x)</b>	<b>685 (6.8 x)</b>	<b>509 (9.2 x)</b>
Multiply-base 2	21,542	513 (42 x)	580 (37 x)	448 (48 x)
Refresh 2	5,508	961 (5.7 x)	617 (8.9 x)	347 (16 x)
<b>Sum</b>	<b>27,050</b>	<b>1,474 (18 x)</b>	<b>1,197 (23 x)</b>	<b>795 (34 x)</b>

Timings (in ms)	CPU Intel DualCore	2x GPU GeForce 260	3x GPU Tesla C1060	4x GPU Tesla C2050
Set-base	14,979	900 (17 x)	715 (21 x)	572 (26 x)
Refresh	5,598	962 (5.8 x)	610 (9.2 x)	347 (16 x)
<b>Sum</b>	<b>20,577</b>	<b>1,862 (11 x)</b>	<b>1,325 (16 x)</b>	<b>919 (22 x)</b>

Timings (in ms)	CPU Intel DualCore	2x GPU GeForce 260	3x GPU Tesla C1060	4x GPU Tesla C2050
Add-base	110,387	1,465 (75 x)	899 (123 x)	872 (127 x)
Refresh	5,621	953 (5.9 x)	608 (9 x)	346 (16 x)
<b>Sum</b>	<b>116,008</b>	<b>2,418 (48 x)</b>	<b>1,507 (77 x)</b>	<b>1,218 (95 x)</b>

# Speed-up factors (compared to CPU)



# Concluding remarks

---

- Top-down planning creates and/or manipulates large numbers of data records
- These updates are **systematic** and **structured**
  - *Involved data points can be enumerated*
  - *Perfectly suited for SIMD-like parallelization on GPUs*
- Increased work compared to CPU algorithm
  - *but pays off in speedup!*
- CUDA implementation up to 95x faster compared to sequential CPU algorithm
- Easy scaling to multiple GPUs