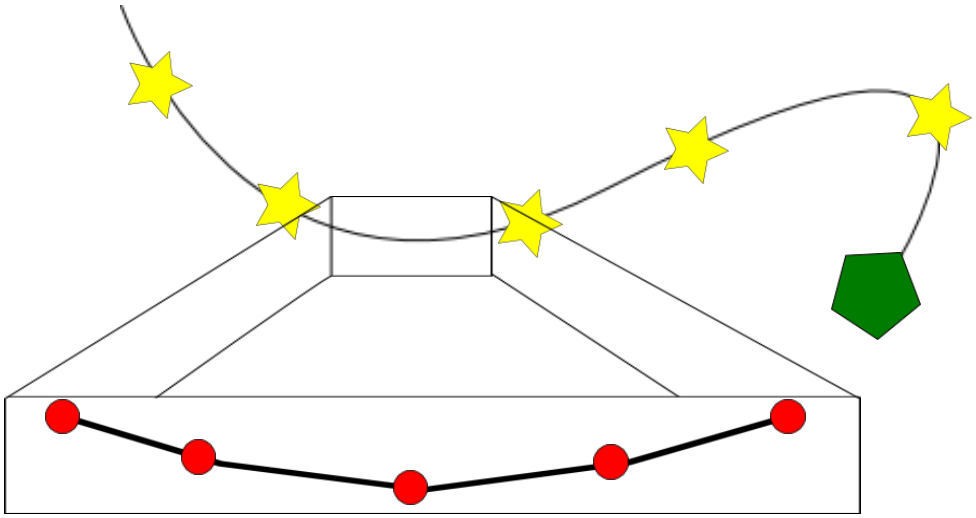
Übung: Parallele Algorithmen mit OpenCL

Sascha Kolodzey

Ablauf

- Lösung und Anregungen zur Trailvisualization
- REDUCTION im Detail
- CONVOLUTION: Ein weiteres paralleles Pattern

Trailvisualisation: Idee



Curve-Vertex

Trail-Particle

Body

Trailvisualisation: Daten Layout

Wir wollen N Bodys mit jeweils L Curve-Vertices und TPpC Partikel simulieren, daraus ergibt sich:

alt:
- boy_Pos (float4, N Stück)
- boy_V (float4, N Stück)

neu:
- curveVertex_Pos (float4, L * N Stück)
  - Spannen die Curves auf
- trailParticle_Pos (float4, TPpC * N Stück)
- trailParticles_S (float, TPpC * N Stück)
  - Interpolationsparameter, global für gesamte Curve eines Partikels
  - Hinweis: 1 zu 1 Beziehung zur Partikel Position
  - Wir könnten den Parameter S auch in der W Komponente der Partikel Position / Velocity speichern

- trailParticles_Dir (float4, N Stück)
  - Nur für die Visualisierung nötig

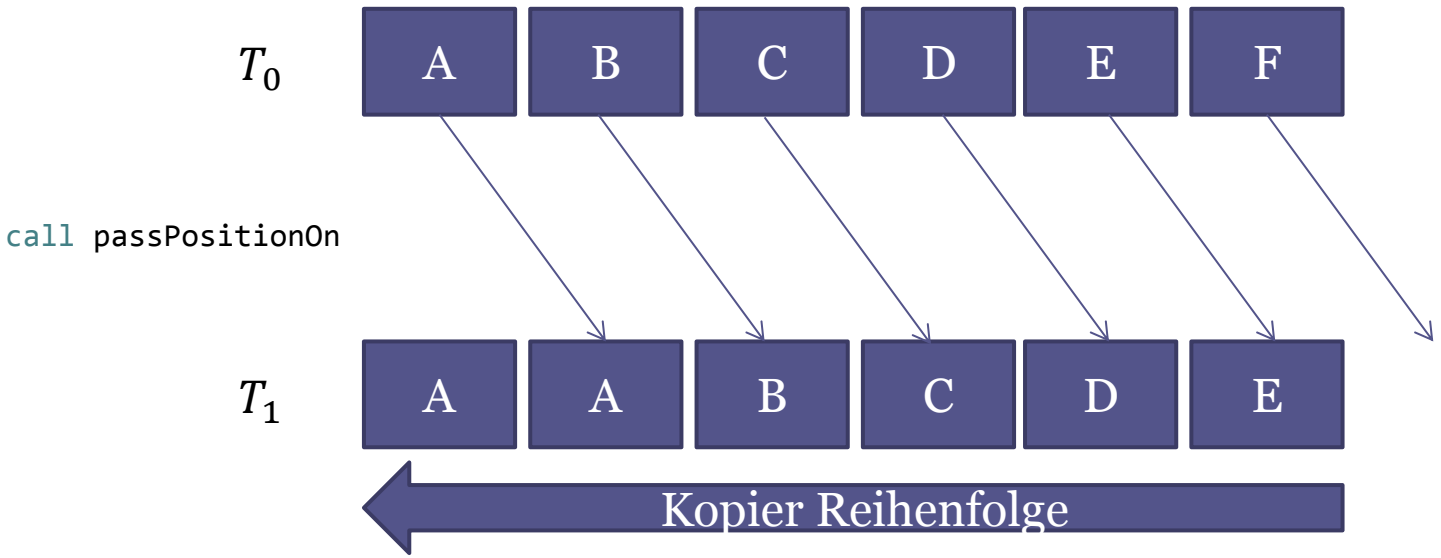
Trailvisualisation: Was war zu tun?

```
While (run)
    call computeNewVs
    call computeNewPs
    call passPositionOn //TODO
    call setTrailParticle //TODO
    call visualize
End
```

Kernel: `passPositionOn`

- 1 Thread pro Curve
- Shifte Partikel-Positionen um einen Index
- Letzte Partikel-Position wird überschrieben
- Es wird eine neue Position frei
- Füge letzte bekannte Position des Bodys ein

Kernel: `passPositionOn`



Achtung: Kopieren erfolgt von „hinten“ nach „vorne“

Kernel: `passPositionOnV2`

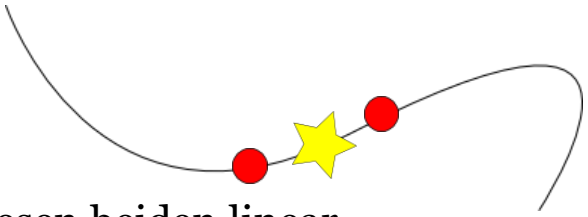
- Momentan 1 Thread pro Curve
- Gibt es eine Möglichkeit die Parallelität zu erhöhen?
- Ja, mögliche Struktur des Algorithmus:
  - Starte $L \cdot N$ Threads mit L als Größe einer Workgroup
  - „Jeder“ Thread läd einen Wert und legt ihn im private Register ab
  - Synchronisation
  - Schreibe geladenen Wert an neue Position
  - Thread mit $localId = 0$ fügt neue Body Position hinzu

Kernel: passPositionOnV2

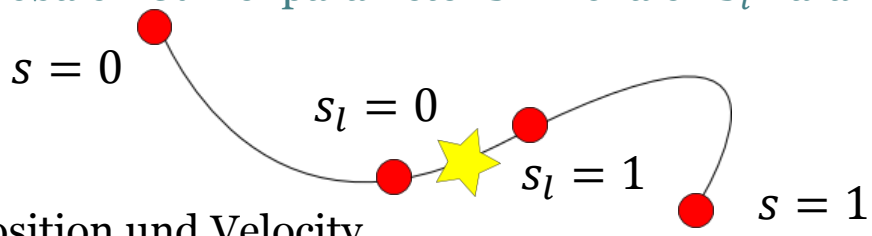
```
kernel void passPositionOnV2(global int* data)
{
    uint id  = get_global_id(0);
    uint lid = get_local_id(0);
    if(lid == get_local_size(0)-1){return;}
    int v = data[id];
    barrier (CLK_GLOBAL_MEM_FENCE);
    data[id+1] = v;
    if(lid == 0) {data[id] = 0;}
}
```

Kernel: `setTrailParticle`

- 1 Thread für jedes Partikel
  - In den Hausaufgaben oft 1 Thread pro Body
  - Problem: Parallelität um den Faktor $TPpC$ eingeschränkt!
- Finde die zwei Curve-Vertices zwischen denen sich das Partikel im Moment befindet



- Interpoliere zwischen diesen beiden linear
  - Dazu nötig: globalen Curvenparameter S in lokalen S_l Parameter umrechnen



- Schreibe neue Position und Velocity
- Velocity ergibt sich aus gelesener Position und neue berechneter Position
- Erhöhe Curvenparameter s um Δs
- Überprüfe ob s größer als 1, wenn ja ziehe 1 ab

Kernel: `setTrailParticle2`

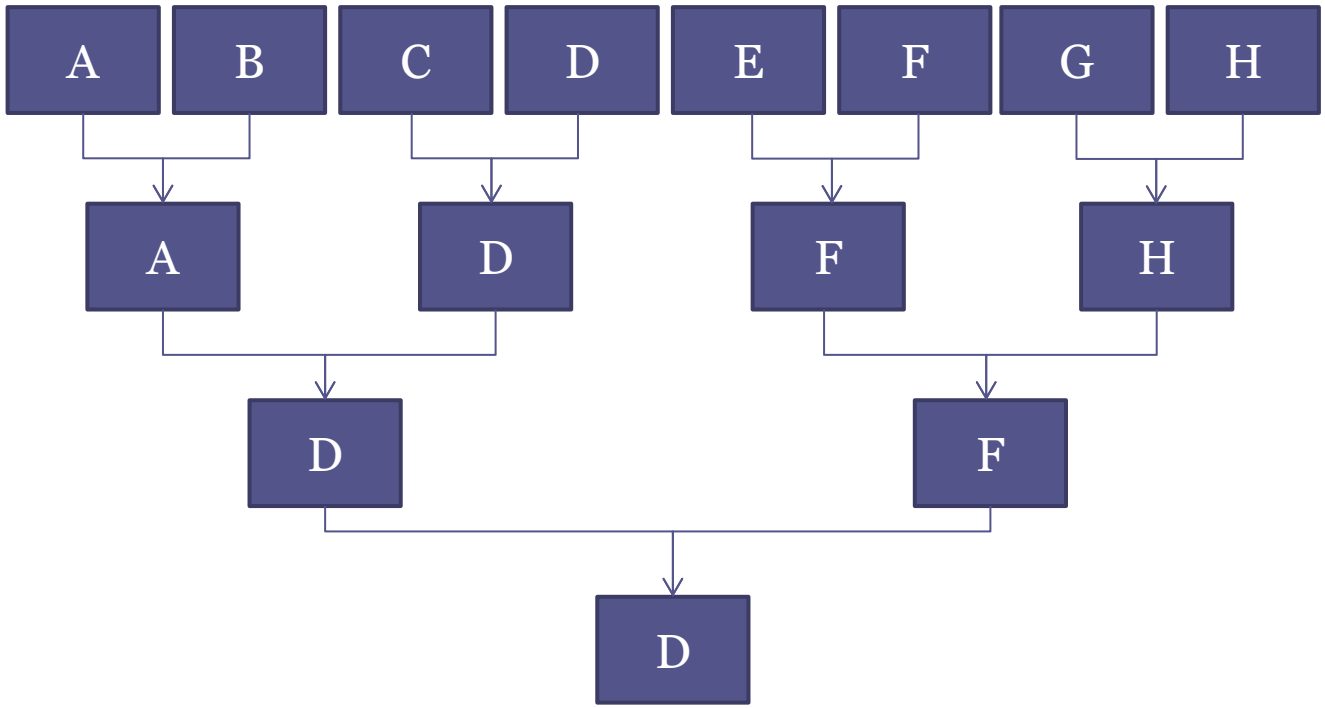
- Weitere Optimierungen mit Synchronisation möglich?
- Für diese Art der Implementation nicht

Ein paralleles Pattern: REDUCTION

- REDUCTION leitet einen einzigen Wert aus einem Array von Werten ab
- „Besuche“ alle Werte und wende Reduktionsoperator an
  - min, max, sum, etc.
- Sequenzieller Ansatz:
  - Durchgehen alle Elemente linear und wende Operator an
- Work Efficient: Jeder Wert wird nur einmal besucht
- Laufzeit: $O(N)$
- Großes N motiviert parallelen Algorithmus

Ein Beispiel: Fußball

- Zweite Runde (K.O. System) der Fußball-Weltmeisterschaft ist eine Maximum-Reduktion



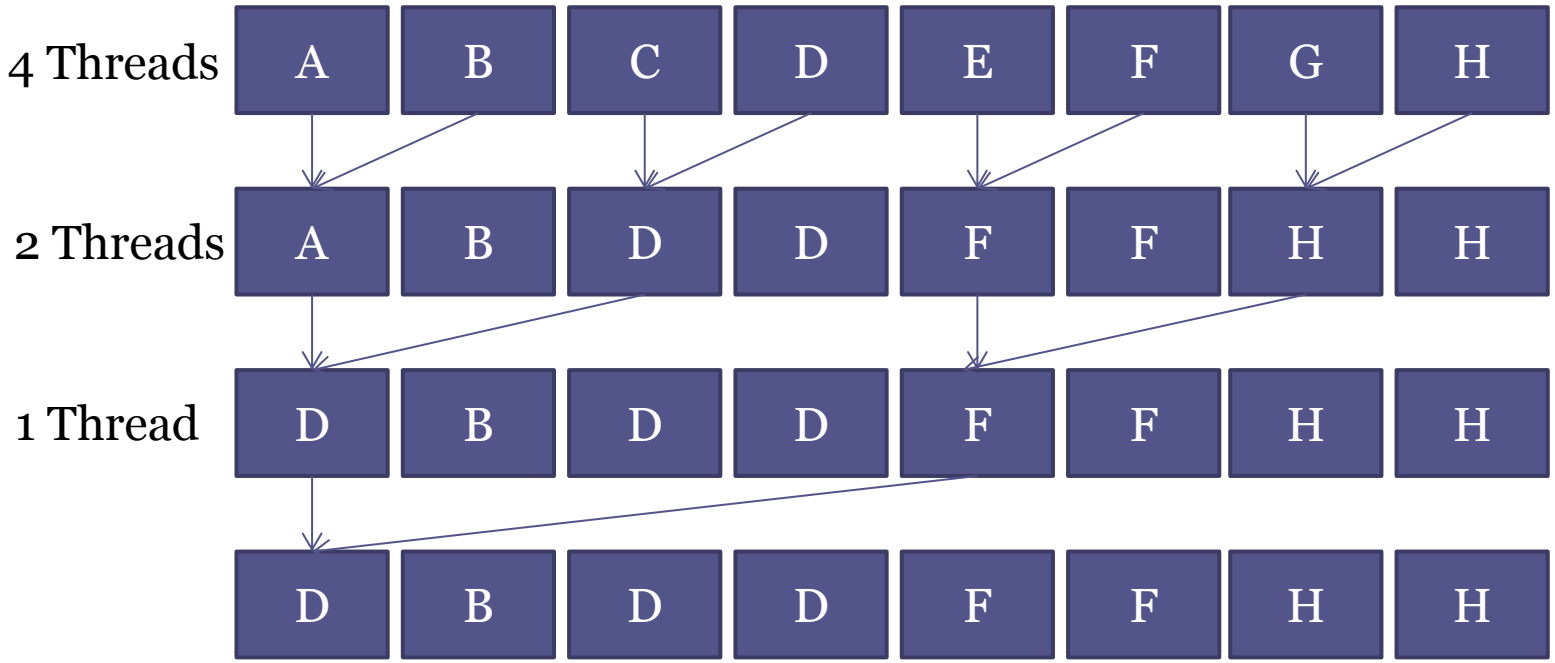
Ein Beispiel: Fußball

- Wir benötigen $\log_2(8)$ Runden um den Gewinner zu ermitteln
- Allgemein: $\log_2(N)$
- Hätten wir 65536 Mannschaften bräuchten wir immer noch nur 16 Runden
- Vorausgesetzt: Für die erste Runde 32768 Fußballplätze und genug Offizielle

REDUCTION ist gar nicht so abstrakt!

REDUCTION: Variante 1

- Start $N/2$ Work-Items für N Daten
- Halbiere die Anzahl der Threads nach jeder Iteration
- $\log_2(N)$ Kernel-Calls



REDUCTION: Variante 1 Recap

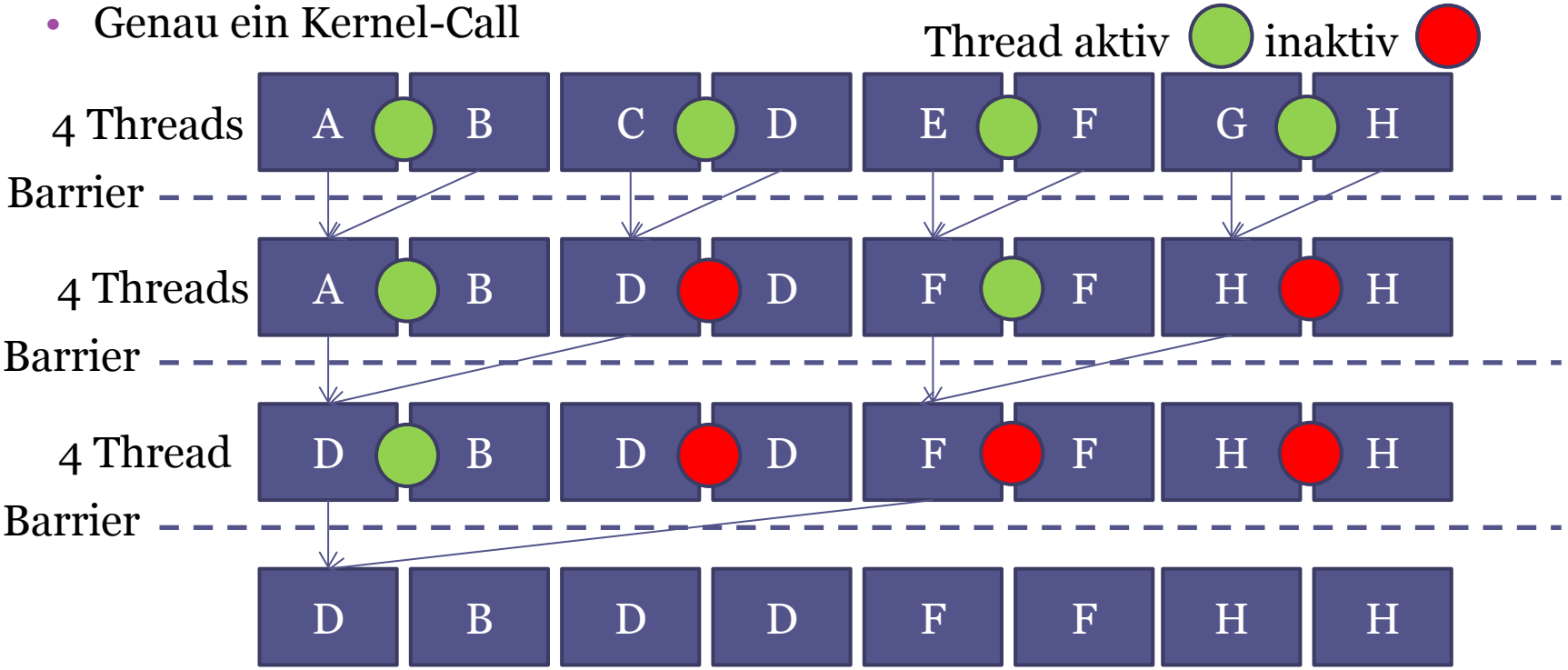
- In der Praxis: Abbruch nicht bei Threadcount 1
- Ab bestimmter Größe Berechnung auf der CPU
- Aus algorithmischer Sicht die beste Lösung
  - Parallelitätsgrad maximal
- Berücksichtigung von Hardware kann jedoch andere Ansätze bevorzugen
- Viele Threads aber i.d.R. 1 FLOP per Operation
- Thread Erzeugung kostet
- Kernel-Call kostet
- Möglicher Ansatz: Lasse einige Threads mehr „Arbeit“ verrichten

REDUCTION: Variante 1 2er Potenzen

- Der Ansatz favorisiert offenbar 2er Potenzen der Datengröße
- Erweiterung für beliebige Inputlänge
- Idee: Sei N die Anzahl der Elemente
  - Starte $(N - N \,\%\, 2) \,/\, 2$ Threads
  - Falls N ungerade war, lasse Thread mit der $ID = get\_global\_size(0) - 1$ drei Elemente reduzieren

REDUCTION: Variante 2

- Synchronisation von Speicherzugriffen innerhalb einer Workgroup möglich
- Idee: Lasse eine Workgroup die gesamte Reduktion durchführen
- Funktioniert so nur mit einer Workgroup
- Genau ein Kernel-Call

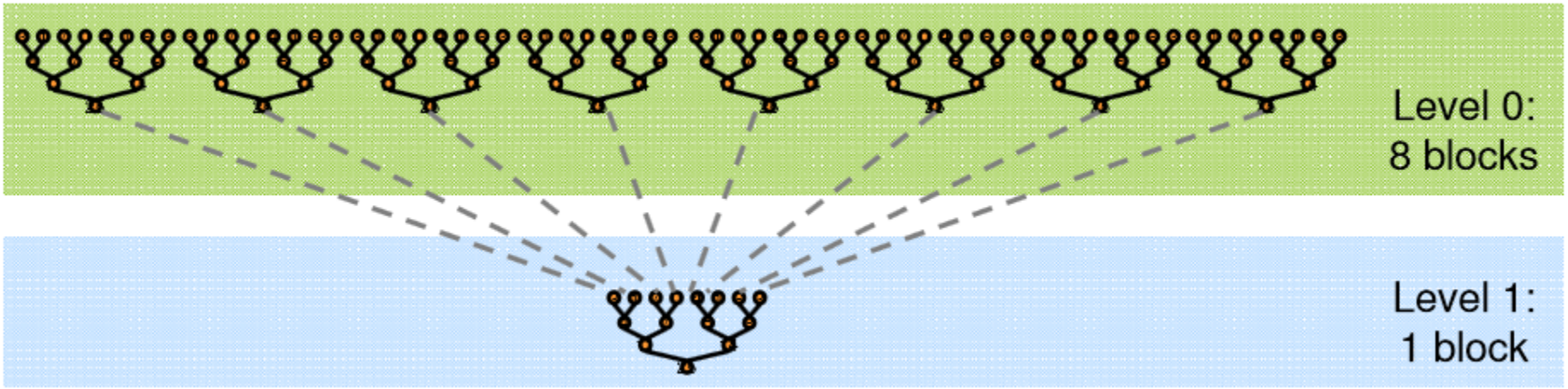


REDUCTION: Variante 2 Reduction Loop

```
for(uint stride = 1; stride <= blockSize; stride <<= 1) {
    if(localthreadId % stride == 0) {
        //Erster Wert bei Index
        int a = 2 * globalThreadId;
        //Zweiter Wer bei Index
        int b = 2 * globalThreadId + s;
    }
}
```

REDUCTION: Variante 3

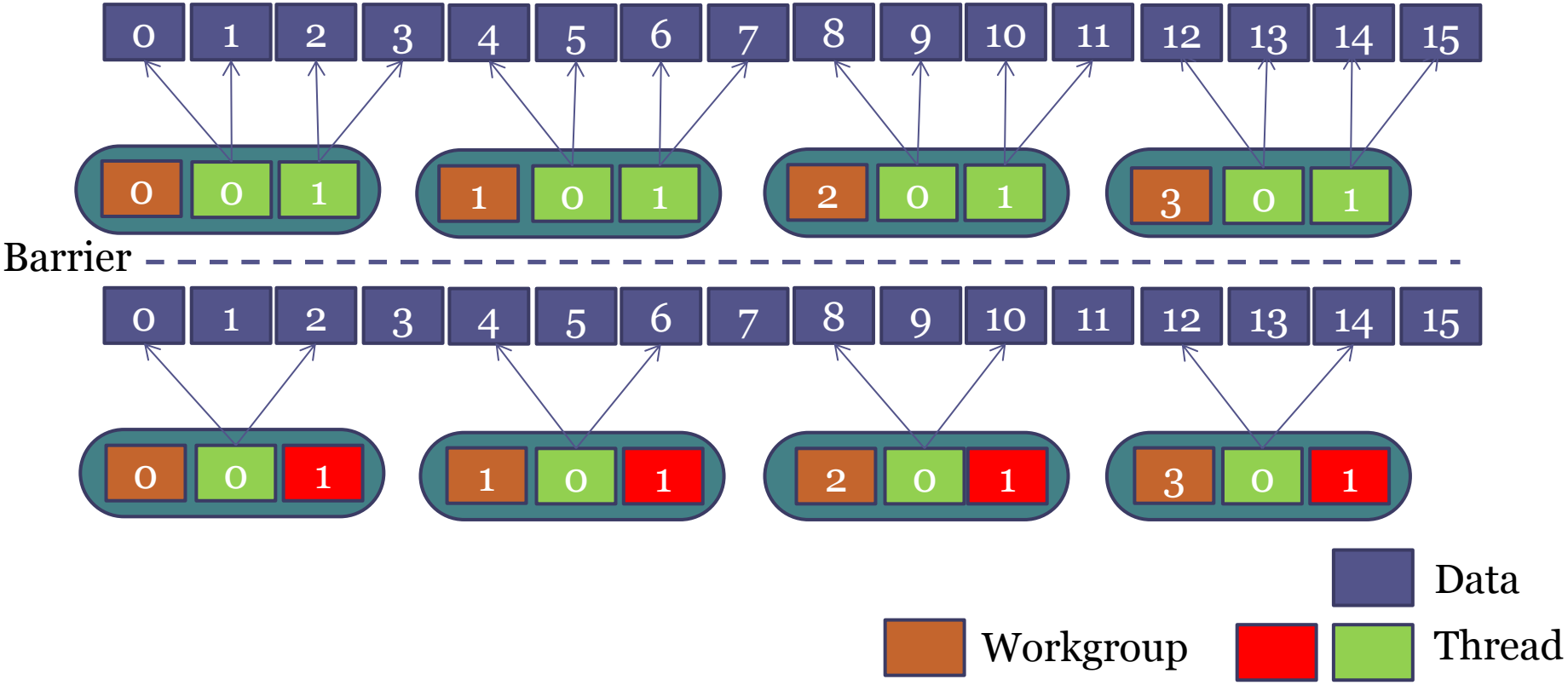
- Erweiterung für Datengrößen die nicht mit einer Workgroup reduziert werden „können“



- Der Code für alle Level ist exakt der gleiche
- Lediglich die Speicherzugriffe müssen neu organisiert sein

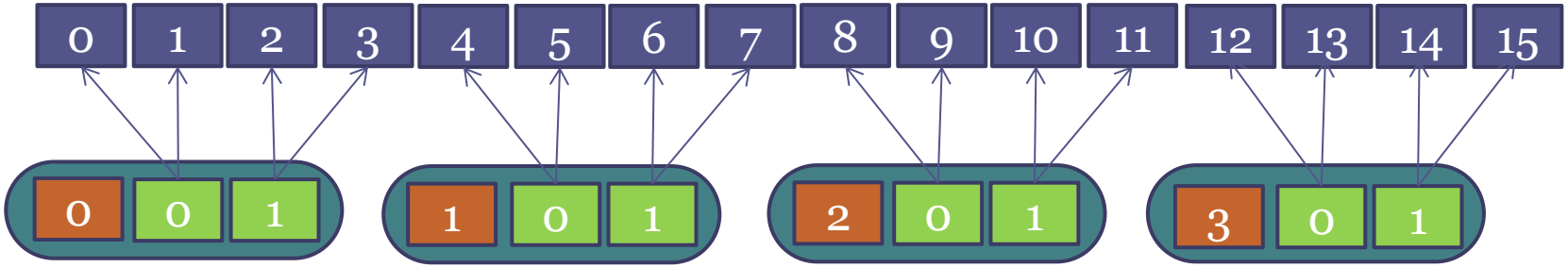
REDUCTION: Variante 3 (Level 0)

- GroupSize= 2, 16 Werte

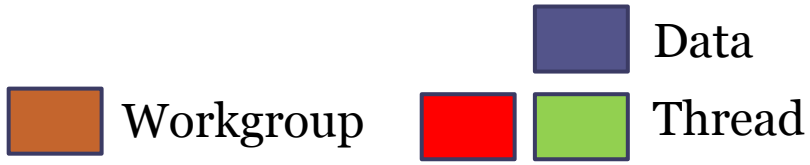


REDUCTION: Variante 3 Speicherzugriffe im Detail (Level 0)

- Erste Loop Iteration

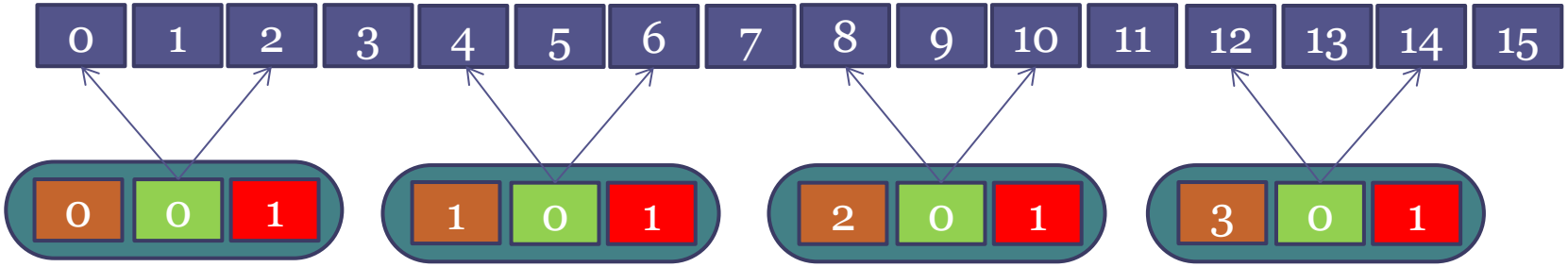


Jeder Thread vergleicht zwei Werte
Erster Wert bei Index:
$2 * (\text{GroupSize} * \text{GroupId} + localThreadId)$ oder $2 * globalThreadId$
Zweiter Wer bei Index:
$2 * (\text{GroupSize} * \text{GroupId} + localThreadId) + 1$ oder $2 * globalThreadId + 1$

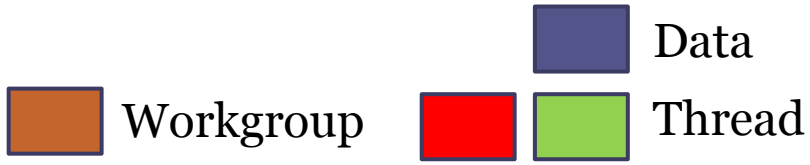


REDUCTION: Variante 3 Speicherzugriffe im Detail (Level 0)

- Zweite Loop Iteration

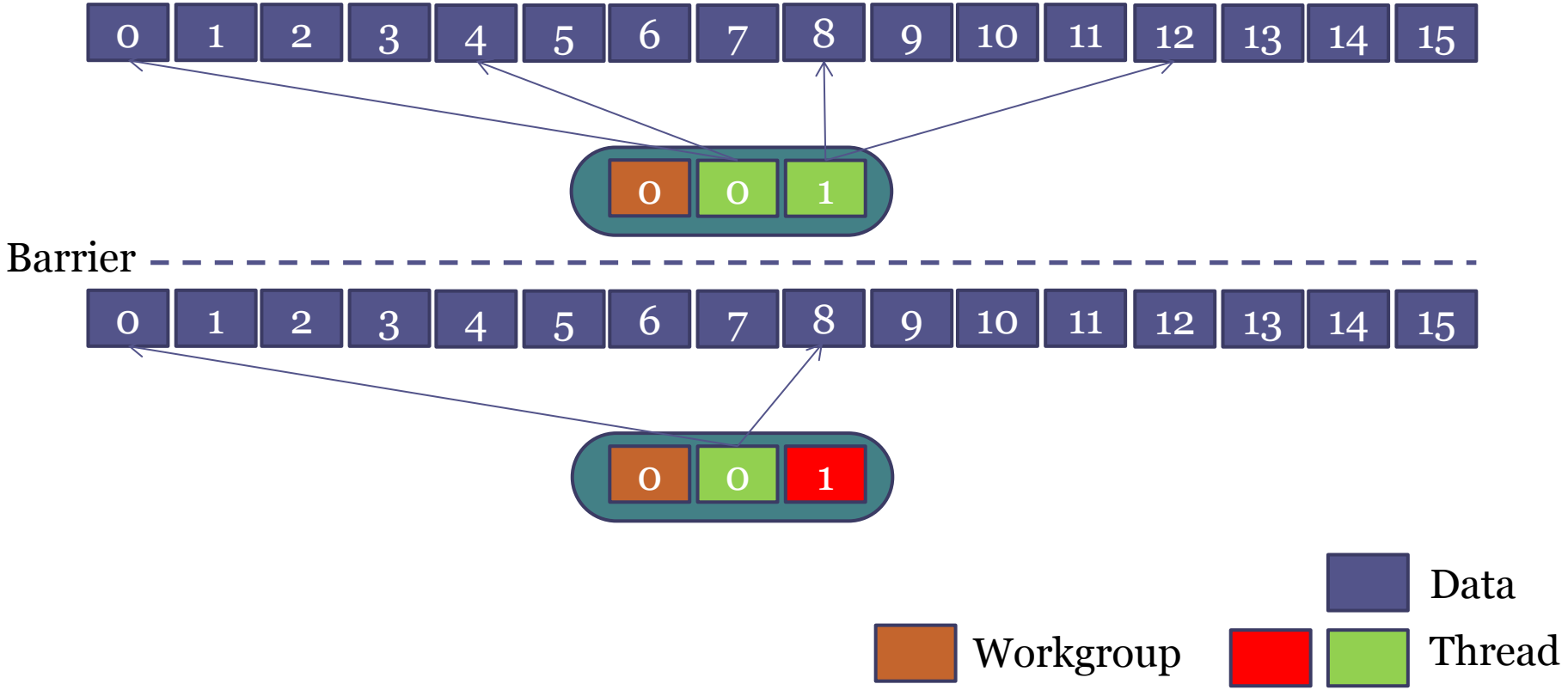


Wenn $localThreadId\ mod\ 2 == 0$ (nur noch die Hälfte der Threads)
Erster Wert bei Index:
$2 * (\text{GroupSize} * \text{GroupId} + localThreadId)$ oder $2 * globalThreadId$
Zweiter Wer bei Index:
$2 * (\text{GroupSize} * \text{GroupId} + localThreadId) + 2$ oder $2 * globalThreadId + 2$



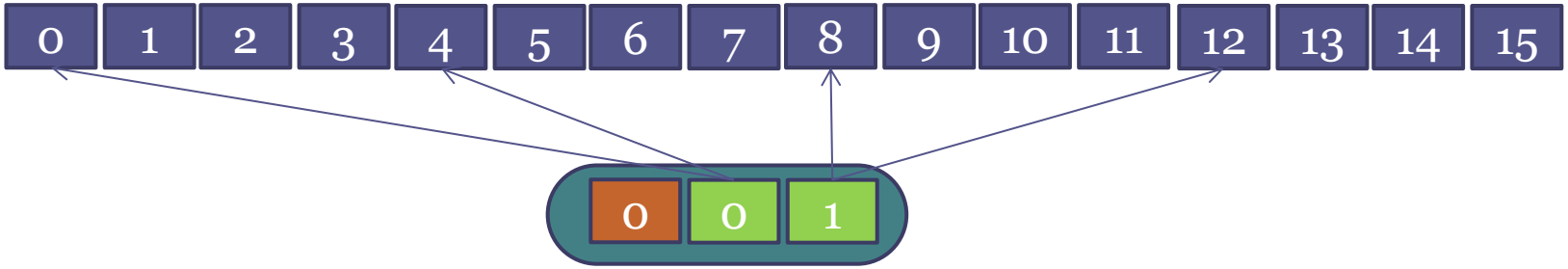
REDUCTION: Variante 3 (Level 1)

- GroupSize= 2, 16 Werte

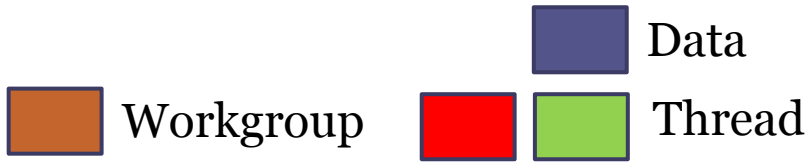


REDUCTION: Variante 3 Speicherzugriffe im Detail (Level 1)

- Erste Loop Iteration

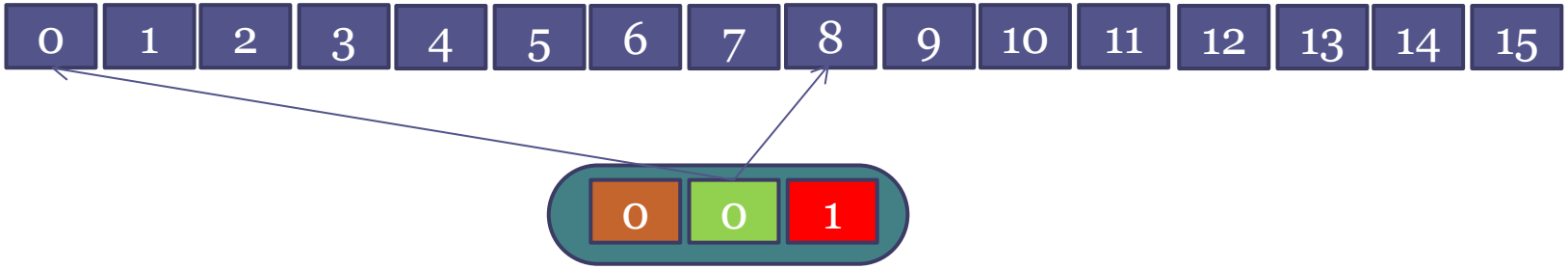


Wir erinnern uns: Im ersten Kernel Call wurden pro Workgroup 4 Elemente reduziert
Also multiplizieren mit 4
Erster Wert bei Index: $4 * 2 * globalThreadId$
Zweiter Wer bei Index: $4 * (2 * globalThreadId + 1)$

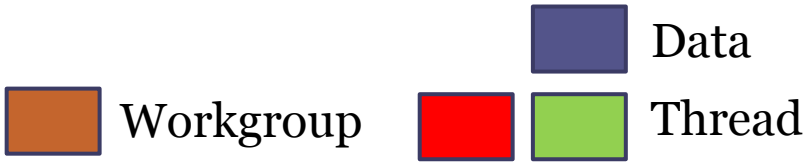


REDUCTION: Variante 3 Speicherzugriffe im Detail (Level 1)

- Zweite Loop Iteration



Wenn $localThreadId \ mod \ 2 == 0$ (nur noch die Hälfte der Threads)
Erster Wert bei Index: $4 * 2 * globalThreadId$
Zweiter Wer bei Index: $4 * (2 * globalThreadId + 2)$



REDUCTION: Variante 3 Kernel Calls

```
workSize <- N / 2
groupSize <- irgendwas < workSize und Teiler von workSize
data <- Folge von Integer Werten
stageScale <- 1
Do(kernelCalls times) {
    call reduce(worksize, groupsize, data, stageScale)
    stageScale <- stageScale * 2 * groupSize
    workSize <- workSize / (2 * groupSize)
    if(workSize < groupSize)
        groupSize = workSize
}
```

- Kernel Calls $= 1 + \log_{2*workgroupSize} \frac{N}{2}$
- Hat dieser Ansatz noch Fehler?
  - Ungerade Anzahl von Elementen
  - Lösung: Wie bei Variante 1

REDUCTION: Variante 3 OpenCL Kernel

```
kernel void reduce(global int* data, const int stageScale) {
    uint id = get_global_id(0);
    uint blockSize = get_local_size(0);
    uint localthreadId = get_local_id(0);
    for(uint s = 1; s <= blockSize; s <<= 1) {
        if(localthreadId % s == 0) {
            int a   = data[stageScale * 2 * id];
            int b   = data[stageScale * (2 * id + s)];
            int res = operation(a, b);
            data[stageScale * 2 * id] = res;
        }
        barrier(CLK_GLOBAL_MEM_FENCE);
    }
}
```

Problem: Gewählte Workgroupsize ist kein ganzzahliger Teiler der Worksize

- Möglichkeiten?
- Versuche algorithmisch eine neue Workgroupsize zu finden
- Finde nächst größeren Teiler und lasse überflüssige Threads nichts tun

Allgemeine Probleme beim Arbeiten mit Workgroups

- Finden der passenden Groupsize für gegebene Datenmenge

```
uint GetGlobalWorkSize(uint dataCnt, uint workSize) {
    if(dataCnt % workSize == 0) {
        return dataCnt;
    }
    return ((dataCnt / workSize) + 1) * workSize;
}
```

- Achtung: Es werden eventuell mehr Threads erzeugt als Daten

Wie sieht es im Kernel aus?

```
kernel void moreThreadsThanData(constant int* src, global int* dst,
uint N) {
    uint id = get_global_id(0);
    if(id >= N) {
        return;
    }
    dst[id] = src[id];
}
```

REDUCTION: Ausblick (GPU)

- Es geht noch besser:
  - Lokaler Speicher
  - Berücksichtigung von Thread-Divergenz
  - Vermeidung von Bank-Konflikten
- Mehr dazu im Kapitel Optimierung

Ein paralleles Pattern: CONVOLUTION

- CONVOLUTION (Faltung) ist eine Array Operation bei der jedes Element eine gewichtete Summer seiner Nachbarn ist
- Anwendungen:
  - Signal-, Image-, Video-Processing
  - Computer Vision
- Die Gewichtungen werden durch eine Input-Maske definiert (Convolution-Kernel / Mask)

CONVOLUTION Beispiel

1D Signal $\{0, 2, 5, 3\}$

1D Convolution Mask $\{\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\}$

Randbedingungen: Repeat

Ergebnis:

$$\{\frac{3}{3} + \frac{0}{3} + \frac{2}{3}, \frac{0}{3} + \frac{2}{3} + \frac{5}{3}, \frac{2}{3} + \frac{5}{3} + \frac{3}{3}, \frac{5}{3} + \frac{3}{3} + \frac{0}{3}\}$$

Mittelwert des Elements und seinen direkten Nachbarn

CONVOLUTION und Constant Memory

- Convolution-Mask ändert sich nicht zur Laufzeit des Kernel
- Speicher Convolution-Mask im Constant Memory
- Im Kernel: *constant* Qualifier statt *global*
- Erstelle Datenbuffer mit *CL_MEM_READ_ONLY*
- Achtung: Speicher Allokierung pro Buffer beschränkt
- Abfrage durch : *CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE*
- Bei GTX 480 65536 Byte (~1600 Floats)

Demos