Parallele Algorithmen mit OpenCL

Universität Osnabrück, Henning Wenke, 2013-05-29

Kapitel

Parallelität

[1]: Parallel Programming (Rauber, Rünger, 2007)

[2]: Algorithms Sequential & Parallel A Unified Approach (Miller, Boxer, 2013)

Parallelität: Grundbegriffe

- Algorithmus
  - **Task**: Parallel lösbares Teilproblem eines Algorithmus. Aufteilung nicht eindeutig
  - Entspricht etwa **Work-Item** in OpenCL
  - **Granularity**: Anzahl der Instruktionen je Task
  - **Potential Parallelism**: Eigenschaft eines Algorithmus. Legt mögliche Arten der Aufteilung in Tasks fest
- Hardware
  - **Thread**: Sequenz von Operationen, die unabhängig abgearbeitet werden kann
  - **Scheduling/Mapping**: Zuordnung der Tasks zu Threads und letztendlich zu Hardware Ressourcen
  - In OpenCL nur indirekt beeinflussbar

Beispiel (Hardware vereinfacht)

- Aufgabe: Skalarprodukt 16-dimensionaler Vektoren (= potential Parallelism)
- Hardware: 4 Threads
- Variante 1: Alle Komponenten Parallel: 16 Tasks
  - Erzeugt 4 Threads und verarbeitet erste 4 Tasks
  - Erzeugt 4 Threads und verarbeitet zweite 4 Tasks
  - Erzeugt 4 Threads und verarbeitet dritte 4 Tasks
  - Erzeugt 4 Threads und verarbeitet vierte 4 Tasks

Thread	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
Komponente	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- Variante 2: Je Thread 4 Komponenten sequentiell: 4 Tasks
  - Erzeugt 4 Threads und verarbeitet alle 4 Tasks
  - Hier: Weniger Threads zu verwalten
  - Oft andere Speicherzugriffsmuster möglich (Variante hier: konsekutiv)
  - Zwischenergebnisse weiterverwenden

Thread	0				1				2				3			
Komponente	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Exkurs: Instruction Level Parallelism

- Folge von Instruktionen kann Parallelität enthalten:

```
1. e ← a + b
2. f ← c + d
3. g ← e + f
```

- Hier: Anweisung 1 und 2 unabhängig. Parallel ausführbar
- Ziel: Identifiziere parallel ausführbare Instruktionen auch in sequentiell formulierten Algorithmen
- Statischer Ansatz: Identifikation & Parallelisierung zur Compile-Zeit durch Software
- Dynamischer Ansatz: Identifikation & Parallelisierung zur Laufzeit durch Hardware
  - Nachteil: Aufwändige Hardware nötig
  - Vorteil: Erreichbarer Parallelitätsgrad höher

Klassifikation nach Flynn

- Instruction-Stream: Sequenz von Instruktionen zur Ausführung durch Computer
- Data-Stream: Sequenz von Daten zur Verarbeitung durch Instruction-Stream
- Flynn [1966] unterscheidet vier Hardware Architekturmodelle:
  - Single-Instruction, Single-Data (SISD)
  - Multiple-Instruction, Single-Data (MISD)
    - „(…) no commercial parallel computer of this type has ever been built“ [1]
    - „(…) is a model that doesn‘t make much sense“ [2]
  - Single-Instruction, Multiple-Data (SIMD)
  - Multiple-Instruction, Multiple-Data (MIMD)

Single-Instruction, Single-Data

- Eine Folge von Anweisungen
- Davon wird eine per Zyklus auf ein Element des Datenstreams angewandt
- Entspricht „von Neumann Modell“
- Algorithmen:
  - Sequentielle Algorithmen
  - Veranstaltung „Info A: Algorithmen“
- Hardware:
  - Intel: Prozessoren bis Pentium
  - Ab Intel Pentium MMX (1995) kein SISD mehr

Single-Instruction, Multiple-Data

- Genau eine Anweisung pro Zyklus …
- … auf unterschiedliche Daten-elemente angewandt
- Parallel & synchron auf allen Prozessoren

Beispiel: Berechne c = a +b, mit a = (1,2) und b = (7, 3) SIMD parallel

```
For each (i | i ∈ {0, 1})
in parallel do

   // Read Elem
   private aL ← a[i]


   // Read Elem
   private bL ← b[i]


   // Execute computation
   private cL ← aL + bL


   // Write back Elem
   c[i] ← cL

end
```

	Global Data		Local Data					
			Processor 1			Processor 2		
Zyklus	c[0]	c[1]	aL	bL	cL	aL	bL	cL
0	/	/	/	/	/	/	/	/
1	/	/	1	/	/	2	/	/
2	/	/	1	7	/	2	3	/
3	/	/	1	7	8	2	3	5
4	8	5	1	7	8	2	3	5

Single-Instruction, Multiple-Data II

- Sobald Algorithmus Bedingungen enthält, liegt kein SIMD vor:

```
In : a[], b[]
Out: c[]
For each (i | i ∈ {0, …, n − 1}) in parallel do
    if(a[i] > b[i]) then
        c[i] ← a[i]
    else
        c[i] ← b[i]
    end
end
```

- Aktuelle SIMD-Hardware kann einzelne Prozessoren dynamisch deaktivieren
- Alle Prozessoren nehmen einmal „if-Pfad“, einmal „else-Pfad“ und verwerfen „falsches“ Ergebnis
- SIMD heute:
  - Viele alte Rastergrafik-Algorithmen (< Shader Model 3) & alte GPUs
  - GPUs: „Fine-grained SIMD“: HD 7970: 64-wide, Nvidia: 32-wide
  - Aktuelle CPUs: SIMD Erweiterungen: SSE, …

Multiple-Instruction, Multiple-Data

- Verschiedene Prozessoren führen unterschiedliche „Programme“ parallel auf verschiedenen Daten aus
- Passiert asynchron
- Hardware Beispiele
  - Mehrkern CPU
  - Cluster
- Anwendung: Schreibprogramm läuft auf einem Kern, Virenscanner auf einem anderen
- Zur Formulierung paralleler Algorithmen unhandlich

Ergänzung zu Flynn: SPMD

- Single program, multiple data
- Ein Programm wird parallel für alle Elemente eines Datenstreams ausgeführt
- Kann Bedingungen enthalten
- Wird asynchron ausgeführt
- Alle Threads haben gleiche Rechte
- Explizite Synchronisation, mit Einschränkungen, möglich
- Spezialfall von MIMD
- Beispiele
  - OpenCL / Cuda Kernel
  - OpenGL / DirectX Shader
  - MPI
  - …

Fragen

- SPMD mit OpenCL möglich?
  - Klar: Ein Kernel & mehr als ein work-item
- MIMD mit OpenCL möglich?
  - Ja: Verschiedene Kernel gleichzeitig ausführen:
  - Unabhängige Queues
  - Out-Of-Order Queue
  - Unterstützt aber nicht jede (einzelne) Hardware
- SIMD mit OpenCL möglich?
  - Nein, da innerhalb eines Kernels globale Synchronisation unmöglich

Einige Arten der Parallelität

- **Task-Parallelism**
  - Entspricht MIMD
- **Data-Parallelism**: Überbegriff für Algorithmen der Kategorien:
  - SIMD
  - MIMD
  - SPMD
  - Im Fokus dieser Veranstaltung
- Weitere, z.B.: **Instruction Level Parallelism**
  - Besprechen wir hier nicht weiter

Beispiel

- Kopiere Elemente eines Datenstreams eine Position weiter

```
Data : a[], n + 1 Elemente, letztes frei
For each (i | i ∈ {0, …, n − 1}) in parallel do

    // Read Elem
    private aLocal ← a[i];

    // Write back Elem
    a[i + 1] ← aLocal

end
```

- Modelle: SIMD ✓ MIMD ✓ SPMD ✓
- Funktioniert?
  - SIMD ✓ da implizit synchron
  - MIMD ✗
  - SPMD ✗

Kapitel

Synchronization

Parallelität: Allgemeines

- Kann nötig sein, wenn Threads mit gemeinsamen Daten arbeiten
- Reihenfolge sicherstellen
  - Threads benötigen Daten, die durch andere Threads zuvor geschrieben werden müssen
  - Threads benötigen Daten, bevor sie durch andere Threads überschrieben werden
  - Unterscheidung: Verwendung Ergebnisse vorheriger Threads (Producer/Consumer) oder Kombination von Zwischenergebnissen
- Sequentielle Ausführung sicherstellen
  - Mehrere Threads verändern die gleichen Daten

Bisherige Beispiele

```
kernel void vec_add(
 global int* a,
 global int* b,
 global int* c){
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

Vektoraddition

- Datenzugriff direkt von work-item Index abhängig
- Keine gemeinsame Datenverwendung
  => Kein Konflikt
- Außerdem: Alle Daten entweder Read-Only oder Write-Only

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

Matrixmultiplikation: C = A * B

- Viele work-items greifen, potentiell parallel, lesend auf gleiche Werte in A und B zu
- Alle Daten entweder Read-Only oder Write-Only
  => Kein Konflikt

Bisherige Beispiele: Nbody System

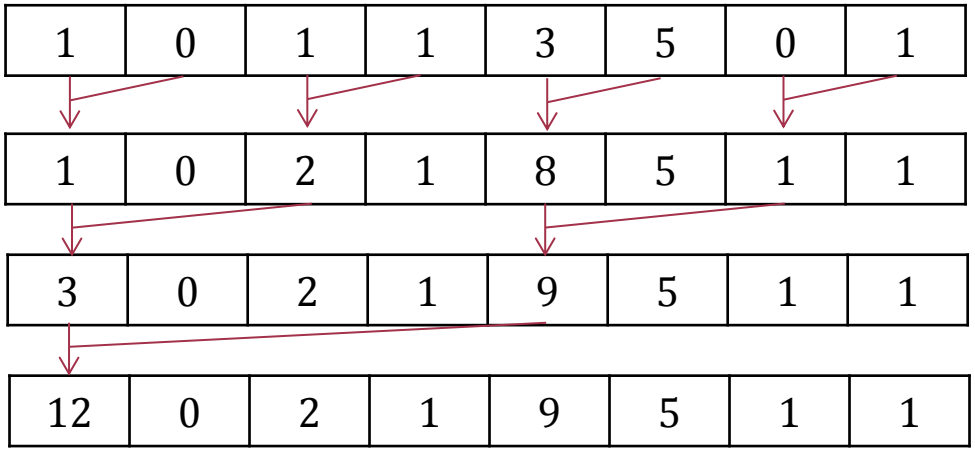
Neue Position eines Simulationsschritts abhängig von Positionen aller Körper

```
// Lösung 1: Umformulierter Algorithmus mit neuem Ort pNeu kommt ohne
//  Synchronisation aus
While (Simulating)
  | For Each (Body b | b ∈ {0, …, N-1}) in parallel do
  |   | // F[b] ← ∑j gravity(p[b], p[j])
  |   | //  ...
  |   | pNeu[b] ← p[b]  + f(p[j, j in {0, ..., N-1}])
  | End
  | toggle(p, pNeu)
End
```

```
// Lösung 2: Aufteilen in 2 Kernel mit implizitem Synchronisationspunkt
// Reihenfolge: Überschreibe p erst, wenn nicht mehr benötigt
While (Simulating)
  | For each (Body b, b in {0, ..., N - 1}) in parallel do
  |   | v[b] ← v[b]  + f(p[j, j in {0, ..., N-1}])
  | End
  | // Global Synchronization Point, alle Threads beendet
  | For each (Body b, b in {0, ..., N - 1}) in parallel do
  |   | p[b] ← p[b] + v[b] · DELTA_T
  | End
End
```

Algorithmus: Reduction (Sum)

- Liefert aggregierten Wert, hier Summe, aller Elemente einer Elementfolge
- Speichert Ergebnis im ersten Element




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

Thread Execution Control

Barrier: Definiert Punkt im Code, an dem jeder Thread warten muss, bis alle Threads ihn erreicht haben

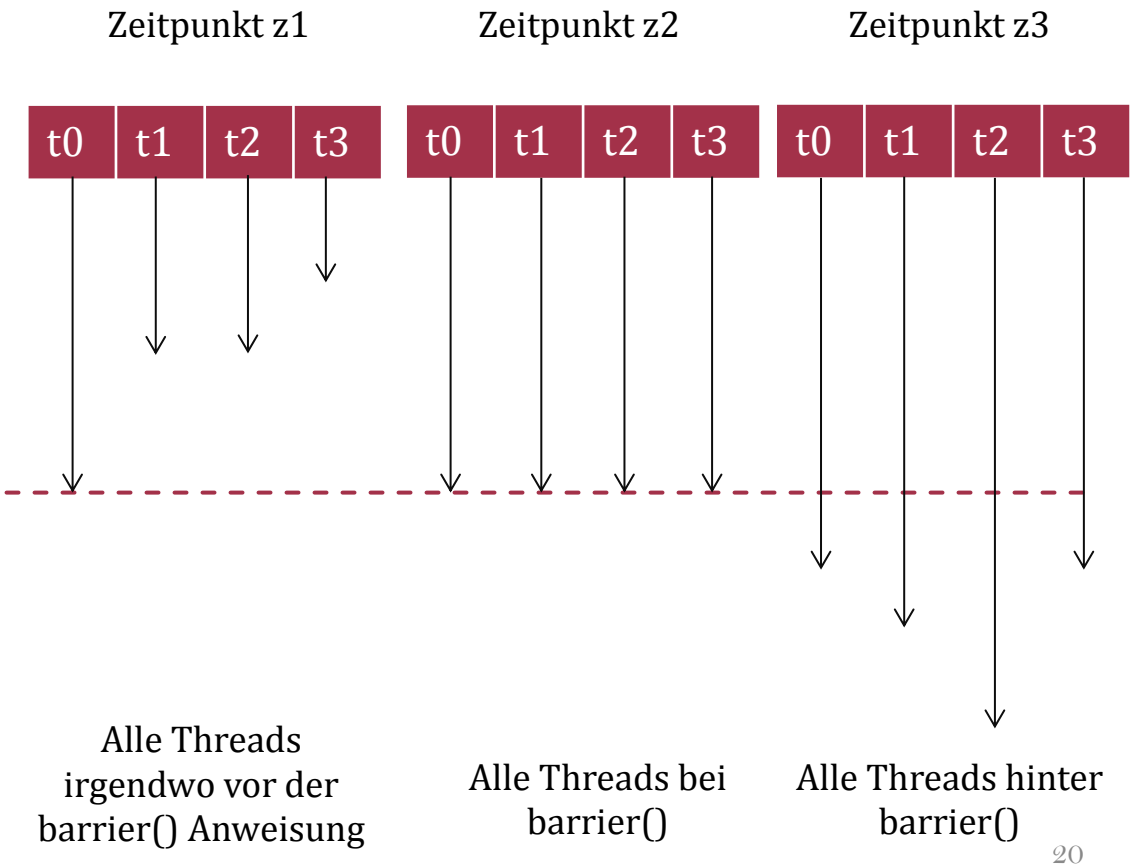
```
For Each (Thread t ∈ {0, …, 3})
in parallel do

   instruction1


   barrier()


   instruction2

End
```



Barrier in OpenCL

- Innerhalb eines Threads in OpenCL nur Work-items einer Work-group synchronisierbar
- Work-groups zur Ausführung auf unabhängigen Compute Units gedacht
- Synchronisation zur Erhaltung der Skalierbarkeit eingeschränkt
- Innerhalb eines Kernels wird Synchronisationspunkt gesetzt mit:

```
// Erst wenn alle work-items einer work-group diese Funktion
//  ausgeführt haben, können work-items fortfahren
void barrier (int flags)

// Werte für flags
CLK_GLOBAL_MEM_FENCE // Stellt Abschluss aller Zugriffe auf globale
                     //  Daten bis zu diesem Punkt sicher

CLK_LOCAL_MEM_FENCE  // Analog für lokale Daten
```

- Achtung: Deadlock, wenn `barrier()` nicht für alle work-items erreichbar

Einschub: Work-Group

- Konsekutive work-items werden immer zu work-groups gleicher Größe zusammengefasst
- Maximale Größe Device-abhängig (Nvidia Fermi: 1024)
- Liefert: `clGetDeviceInfo` für `CL_DEVICE_MAX_WORK_GROUP_SIZE`
- Kann durch Host explizit gesetzt werden mit:

```
int clEnqueueNDRangeKernel(
  int work_dim,     // Dimension der Kernel-Indizierung

  // Globale Anzahl Work Items pro Dim
  PointerBuffer global_work_size,

  // Elementzahl je Work-group je Dimension. global_work_size muss
  //  in jeder Dimension ganzzahliges Vielfaches davon sein.
  PointerBuffer local_work_size, // null: OpenCL entscheiden lassen
  ...
)
```

Einschub: Work-Group (2)

- Work-items erhalten einen innerhalb der work-group lokal eindeutigen Index
- Zugehörige built-in Funktionen:

```
// Liefert lokalen Index in der jeweiligen Index-Dimension D,
//  D ∈ (0, 1, 2)
int get_local_id(int D)     // 0, ..., local_work_size[D]-1

int get_group_id(int D)     // Liefert aufsteigenden Index der
                            //  work-group. 0, …, work-group-cnt-1

int get_num_groups(int D)   // Anzahl der work-groups

int get_local_size(int D)   // Anzahl der work-items je work-group
```

Beispiel

```
clEnqueueNDRangeKernel(
   work_dim <- 1,
   global_work_size <- {12},
   local_work_size <- {3},
   ...
)
```

Erzeugt:

- 12 Work-items
- Gruppiert zu work-groups aus je 3 work-items
  => 12/3 = 4 work-groups

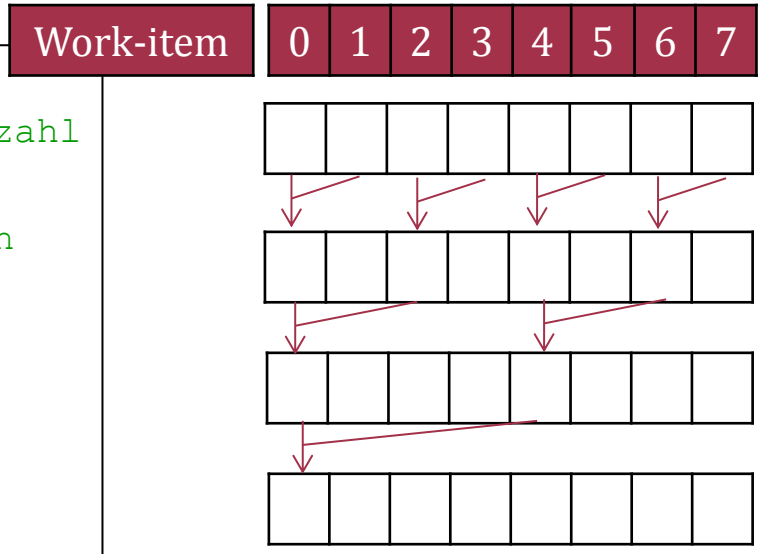
Synchronisierbar jeweils:

Work-item	A	B	C	D	E	F	G	H	I	J	K	L
get_global_id(0)	0	1	2	3	4	5	6	7	8	9	10	11
get_local_id(0)	0	1	2	0	1	2	0	1	2	0	1	2
get_group_id(0)	0			1			2			3		

Reduction mit Barrier im Kernel (naiv)

- Gegeben: `vals`, Folge n Werten , n sei zweier Potenz
- Dann kann Kernel reduction_Sum die Summe berechnen
- Eigenschaften / Bedingungen:
  - Es darf nur eine(!) work-group geben
    => local_work_size = global_work_size
  - Der Kernel wird genau einmal gestartet
  - Es gibt n work-items, davon ≥ n/2 untätig

```
kernel void reduction_Sum(global int* vals,
 const int log2n // Zweierlogarithmus d. Elementzahl
){
  int id = get_local_id(0); // globalId identisch
  int stride = 1;
  for(int i = 0; i < log2n; i++){
     if(id % (2 * stride) == 0)
        vals[id] = vals[id] + vals[id + stride];
     barrier(CLK_GLOBAL_MEM_FENCE);
     stride *= 2;
  }
}
```

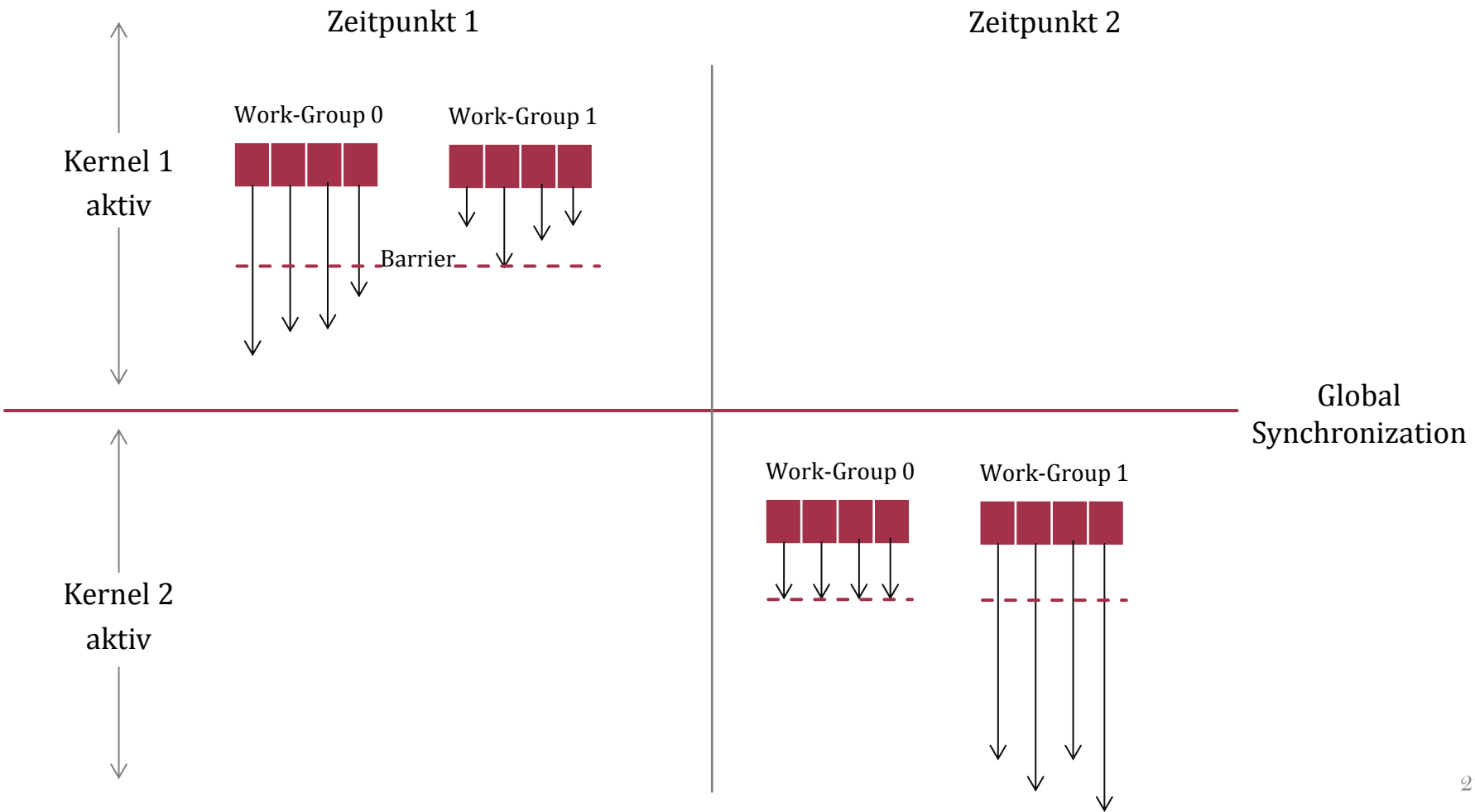


OpenCL Barrier Synchronization

- Device-Side Synchronization: Lokal
  - Innerhalb eines Work-Items werden Speicherzugriffe in der gegebenen Reihenfolge ausgeführt
  - Für Work-Items einer Work-Group sind Speicherzugriffe nur bei Barrier-Anweisungen konsistent
  - Für Work-Items verschiedener Work-Group sind Speicherkonsistenz nicht garantiert werden, bis der Kernel terminiert
- Host-Side Synchronization: Global
  - Zwischen vollständig ausgeführten Kerneln
  - Eine in-order Queue: Implizit
  - Mehrere Queues eines Context und/oder out-of-order Q: Events
  - Sonst: Manuell (`clFinish`)

Synchronisation mehrerer Work-Groups

- Zwei Kernel mit je zwei work-groups a 4 work-items
- Ausgeführt in in-order-queue



Ausblick: Atomic Operations

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