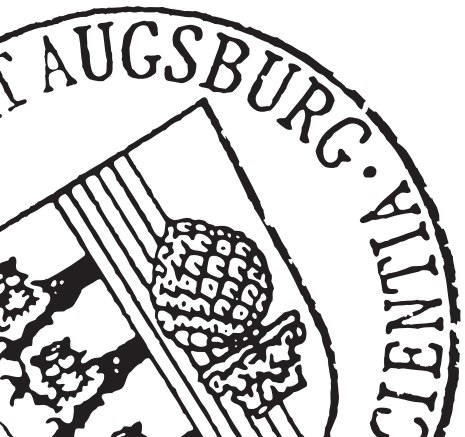
Seminar
Safety-Critical Systems

Universität Augsburg University

Speicherkonsistenzmodelle für GPUs

Marc Blickle



Lehrstuhl für Systemnahe Informatik
und Kommunikationssysteme
Institut für Informatik
Universität Augsburg

2. Februar 2019

Motivation

- Vermehrt datenparallele Entwicklung bei GPU
- Speicherzugriffe auf Shared Memory müssen abgestimmt sein
- Bei CPU etabliert
- GPU-Hersteller hingegen bieten schlechte Transparenz und Info
- Datenparallele Entwicklung wird erschwert

- Was sind etablierte CPU-Speicherkonsistenzmodelle?
- Wie gut oder schlecht sind sie für den GPU-Gebrauch?
- Neue Ansätze?

Gliederung

Grundlagen

Untersuchte Speicherkonsistenzmodelle

Gegenüberstellung

Andere Ansätze

Fazit

Grundlagen

Untersuchte Speicherkonsistenzmodelle

Gegenüberstellung

Andere Ansätze

Fazit

- GPU-Architektur
- Speicherkonsistenz
- Speicherkonsistenzmodelle

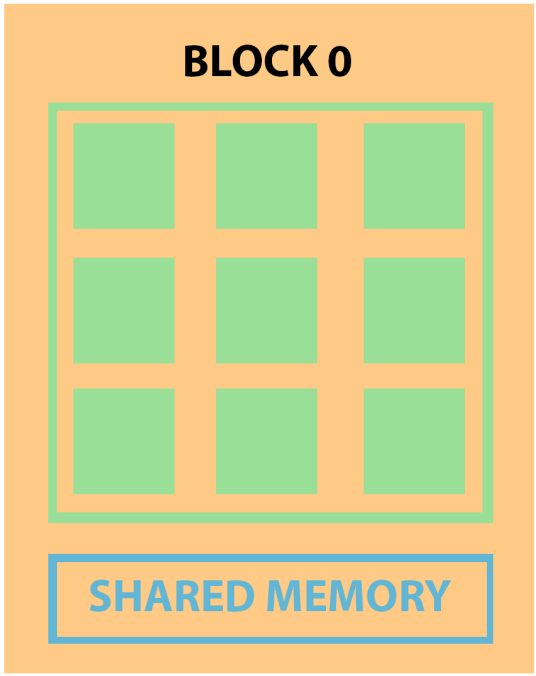
GPU Architektur - ein Thread

- Ist eine einzelne Recheneinheit
- Führt Rechenoperationen aus
- Interagiert mit Speicher
- Mehrere tausend Threads

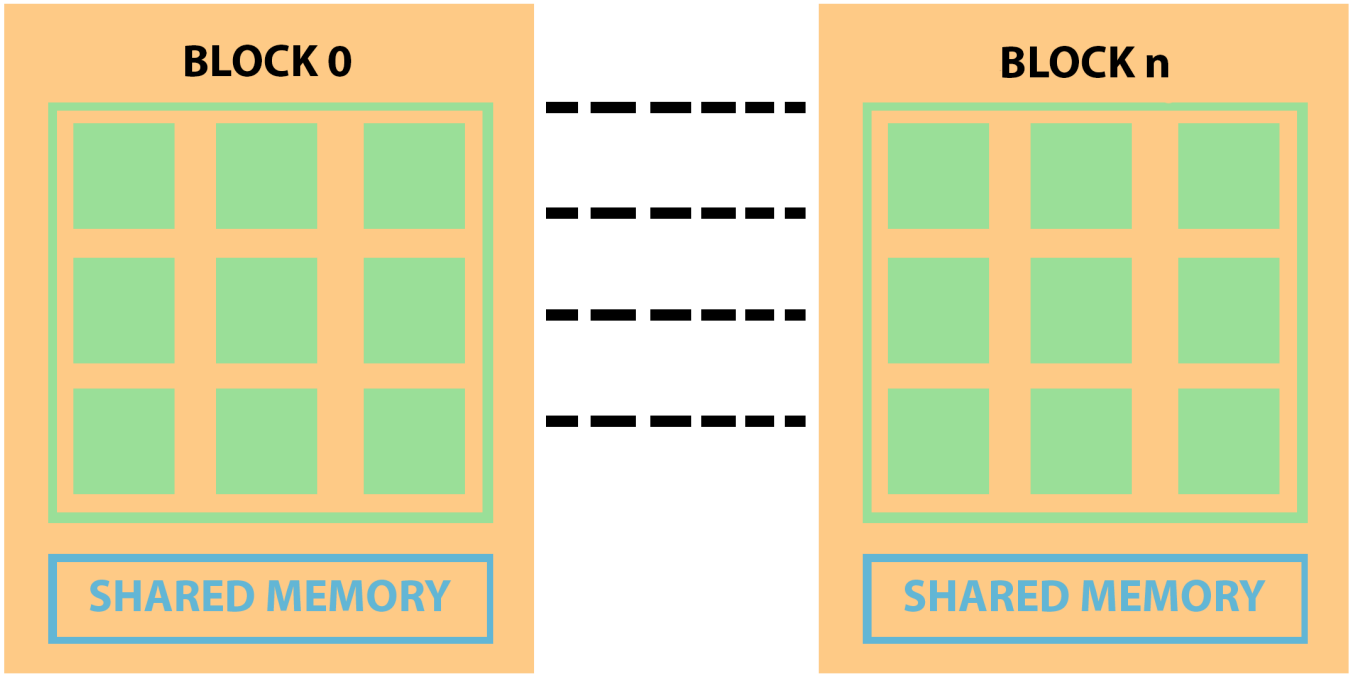
GPU Architektur - ein Block

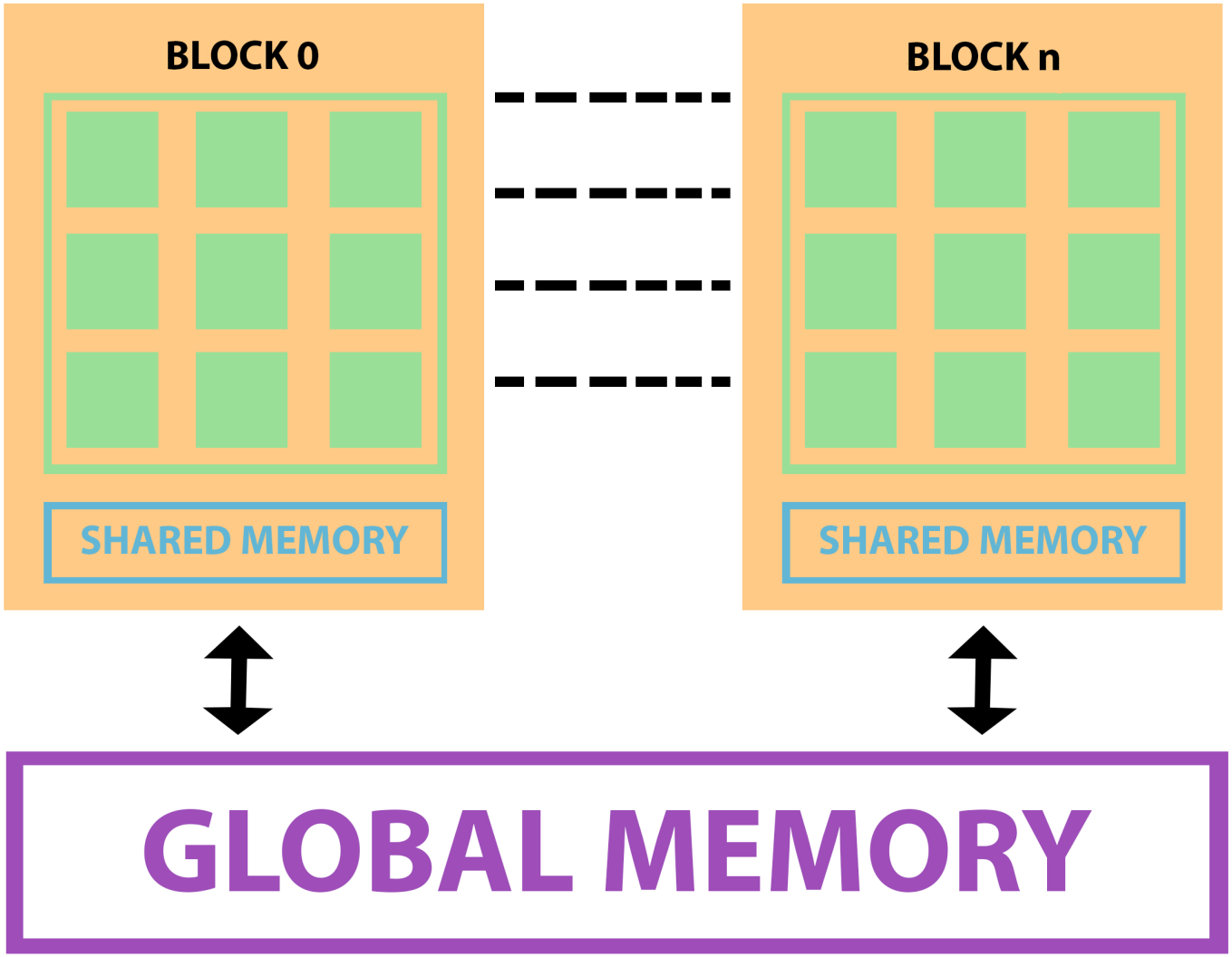
- Mehrere Threads sind in einem Block zusammengefasst
- Anzahl unterscheidet sich von Architektur zu Architektur
- Parallele und oder serielle Ausführung
- Threads in einem Block teilen Shared Memory
- Zum Schreiben und Lesen: Über Shared Memory kommunizieren die Threads

GPU Architektur - ein Block



GPU Architektur




BLOCK 0
BLOCK n
SHARED MEMORY
SHARED MEMORY
GLOBAL MEMORY

- Programmierer entscheidet über Thread- und Blockzahl
- Obergrenze allerdings von GPU vorgegeben
- Wichtig dabei: Threads müssen Speicherzugriffregeln einhalten

Speicherkonsistenz und Speicherkonsistenzmodelle

Speicherkonsistenz

- Threads greifen also gemeinsam auf Shared Memory zu
- Was passiert, wenn mehrere Threads auf selbe Speicheradresse zugreifen wollen?

Speicherkonsistenz

```
boolean P1KS, P2KS

prozess P1 {
  if(P2KS == false){
      P1KS = true;
      Kritische_Operationen;
      P1KS = false;
  }
}
```

```
prozess P2 {
  if(P1KS == false){
      P2KS = true;
      Kritische_Operationen;
      P2KS = false;
  }
}
```

Speicherkonsistenz

```
boolean flag1, flag2, turn;

prozess P1 {
  flag1 = true;
  while(flag2 == true) {
      if(turn != false) {
          flag1 = false;
          while(turn != false) {}
          flag1 = true;
      }
  }

  Kritische_Operationen;
  turn = true;
  flag1 = false;
}
```

```
prozess P2 {
  flag2 = true;
  while(flag1 == true) {
      if(turn != true) {
          flag2 = false;
          while(turn != true) {}
          flag2 = true;
      }
  }

  Kritische_Operationen;
  turn = false;
  flag2 = false;
}
```

Speicherkonsistenz

- Speicherdaten im System sollen einheitlich und widerspruchfrei sein
- Es werden Anforderungen geschaffen, die erfüllt werden müssen um dies zu erreichen
- Diese bestimmen, was eine Operation sehen und machen darf
- Verschiedene Architekturen bieten verschiedene Anforderungen: verschiedene Speicherkonsistenzmodelle

Speicherkonsistenzmodelle

- Modell stellt Vertrag zwischen Software und Speicherhardware dar
- Regelwerk, wie das System Speicheroperationen verarbeitet
- Bei Einhaltung werden Ergebnisse zugesichert und konsistente Ziele erreicht
- Es herrscht dann Speicherkonsistenz
- Verschiedene Modelle - verschiedene Strukturen - verschiedene Zwecke

Welches Modell ist nun gut für eine GPU?

Welches Modell ist nun gut für eine GPU?

- GPU sehr viele Recheneinheiten
- GPU verfügt über tieferen Aufbau

- Sehr viele parallele Ausführungen

→ GPU braucht Schnelligkeit und Performance, viele Berechnungen
→ Speicher soll geregelt beschrieben und gelesen werden

→ Aber Threads sollen dies schnell und ungehindert tun

Gliederung

Grundlagen

Untersuchte Speicherkonsistenzmodelle

Gegenüberstellung

Andere Ansätze

Fazit

Untersuchte Speicherkonsistenzmodelle

- Sequential Consistency Model
- Weak Consistency Model
- Release Consistency Model

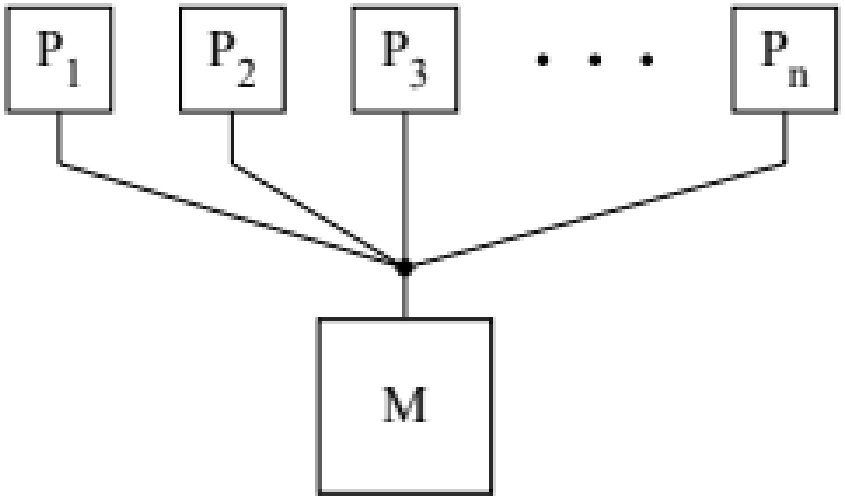
Sequential Consistency Model

Sequential Consistency Model

Abstrakt:

- Alle Threads über Switch an gemeinsamen Speicher gekoppelt
- Switch gibt Speicher für bestimmten Prozessor frei
- Dieser darf nun operieren, meldet wenn fertig
- Switch sortiert also die Lese und Schreiboperationen und stellt sequentielle Reihenfolge her
- Unabhängig davon, wann sie eingereicht werden, sortiert Switch sinnvoll
- Alle Prozessoren sehen somit dieselbe Reihenfolge

Sequential Consistency Model



Program Order

R R W W
| | | |
R W R W

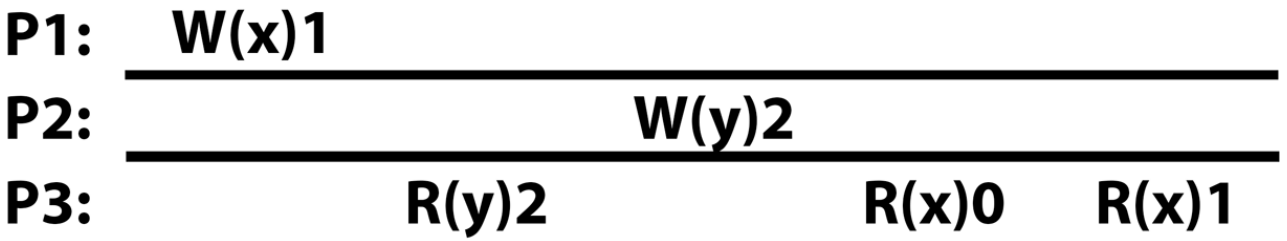
Sequential Consistency Model

Konsistenz resultiert aus Programm, wenn:

- Jeder Prozessor Speicheranfragen in der Reihenfolge einreicht, die Programm vorgibt
- Speicheranfragen jedes Prozessors von Schlange verarbeitet werden, die die Anfragen sortiert

Globale Reihenfolge für alle eingereichten Operationen!

Sequential Consistency Model



Sequential Consistency Model

Vorteile:

- Ablauf ähnelt stark der Denkweise eines Programmierers

Nachteile:

- Leider ist dieser nicht parallel
- Modell passt nicht wirklich zu Multiprozessorsystem
- Sequentielle Abfolge verlangsamt die Ausführung
- Je mehr Prozessoren, desto schlechter für GPU
- Prozessoren müssen immer aufeinander warten

Schwaches Speicherkonsistenzmodell

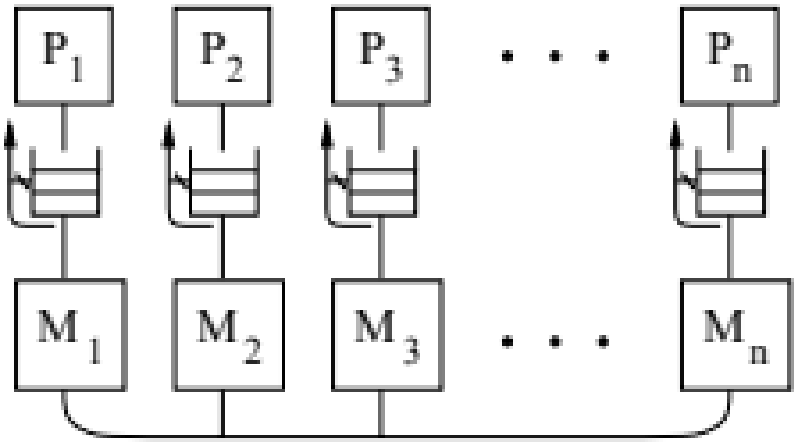
Weak Consistency Model

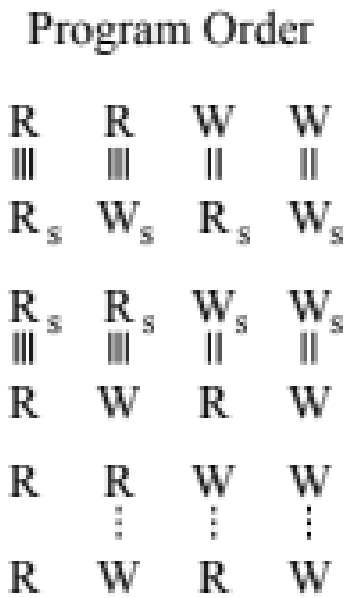
- Zu dem Zweck entwickelt, Speicherlatenz zu verringern
- Weiterentwicklung aus sequentiellem Modell
- Leistungssteigerung von 40%

Weak Consistency Model

- Datenoperationen:
  - Werte werden aus Speicher gelesen oder geschrieben
  - Diese Operationen dürfen parallel zueinander ablaufen
- Deshalb Synchronisationsoperationen:
  - Bringt Shared Memory auf einen Stand
  - Jeder Prozessor kann diese auslösen
  - Keine neuen Operationen dürfen dann begonnen und laufende müssen abgeschlossen werden
  - Synchronisation ist auch Neuanordnung von laufenden Operationen
  - Optimale Reihenfolge generieren und Performance steigern

Weak Consistency Model





Weak Consistency Model

Schwache Konsistenz ist gegeben, wenn:

- Wenn die Synchronisationsoperationen untereinander sequentiell konsistent sind
- Alle anderen Operation können in beliebiger Reihenfolge gesehen werden
- Alle Prozessoren stets alle Synchronisationsoperationen in gleicher sequentieller Reihenfolge sehen

Weak Consistency Model

Vorteile:

- Einzelne Threads können sich in beliebiger Reihenfolge Zugriff verschaffen
- Müssen nicht auf andere Prozesse warten
- Speicherlatenz wird verringert, Performance gesteigert

Nachteile:

- Aufwändigere Programmierung, umfangreichere Regeln
- Fehleranfälliger
- Bei zuviel Synchronisation kann Performancegewinn zunichte gemacht werden

Release Consistency Model

Release Consistency Model

- Weiterentwicklung des schwachen Modells
- Sychronisationsoperation wird weiter unterteilt

Zweck:
Synchronisationsoperationen sollen nicht mehr warten müssen
Read auf selbe Adresse immer möglich

Release Consistency Model

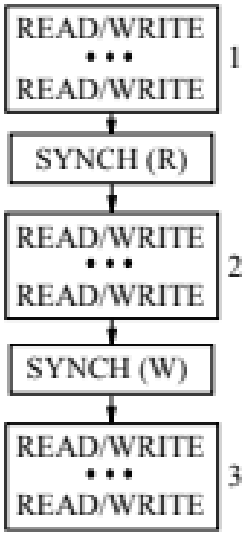
Aquire:

- Überprüft nur, ob alle Schreiboperationen auf den gemeinsamen Dateien fertig
- Falls ja, erhält der 'aquire'-ausführende Prozessor Zugriff
- Funktion zum Anfragen von Berechtigungen

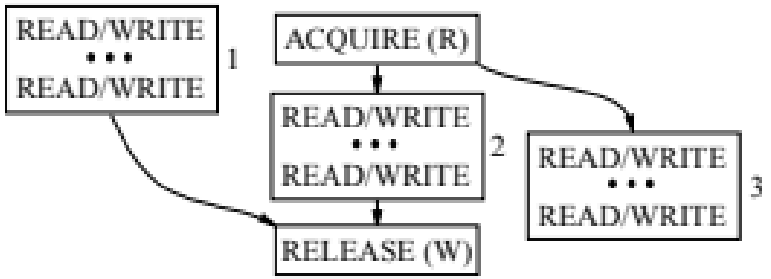
Release:

- Prozessor kann Schreiboperationen zugänglich machen
- Muss allerdings noch nicht mit schreiben fertig sein
- Funktion zum Gewähren von Berechtigungen

Release Consistency Model



(a) Weak Ordering (WO)



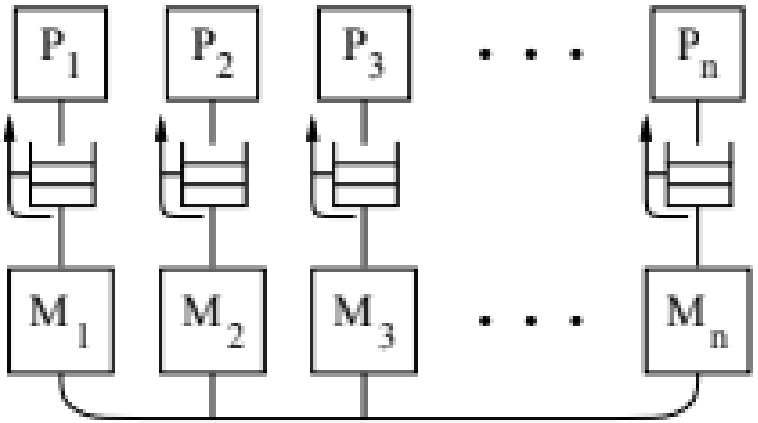
(b) Release Consistency (RC)

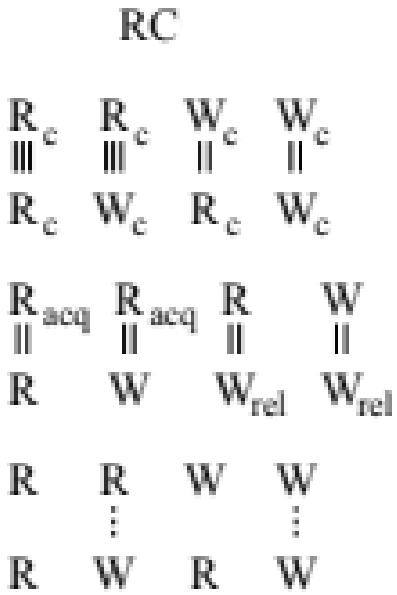
Release Consistency Model

Release Consistency ist sichergestellt, wenn:

- Alle 'aquire'-Operationen durchlaufen sind, bevor geschrieben oder gelesen wird
- Alle Schreib- oder Leseoperationen durchlaufen sind, bevor 'release' startet
- Alle Schreiboperationen von A von B gesehen werden, nachdem A sie released hat und bevor B sie aquired hat

Release Consistency Model





Release Consistency Model

- Release muss auf vorangegange Operationen warten
- Aquire muss nicht auf vorangegangene Operationen warten
- Folgende Operationen warten nicht auf Fertigstellung von Release
- Folgende Operationen warten auf Fertigstellung von Aquire

Release Consistency Model

- Mehr Flexibilität und Schnelligkeit zwischen 'acquire' und 'release'
- Operationen überlappbar
- Es gibt insgesamt mehr Operationen
- Mehr Operationen müssen ausgeführt und geplant werden

Gliederung

Grundlagen

Untersuchte Speicherkonsistenzmodelle

Gegenüberstellung

Andere Ansätze

Fazit

Gegenüberstellung

	Vorteile	Nachteile	GPU-geeignet?
Sequentielles Modell	- Straight forward - Einfach umzusetzen -Kein Synchronisationsaufwand	- Langsam	Nein
Schwaches Modell	- Parallele Ausführung von Operationen - Schnell	- gewisser Synchronisationsaufwand - Entwickler muss genau abstimmen, wann synchronisiert wird	Ja
RC Modell	- Überlappende Ausführung von Operationen mit gleicher Speicheradresse - Schnell	- mehr Rechenleistung für gleiche Arbeit nötig - Komplizierter	Ja

Gliederung

Grundlagen

Untersuchte Speicherkonsistenzmodelle

Gegenüberstellung

Andere Ansätze

Fazit

CUDA und OpenCL

- GPUPU Programmiermodelle
- Fokus auf Optimierung datenparalleler Entwicklung
- Mehrere Threads sollen auf dieselbe Speicheradresse zugreifen können
- CUDA basiert auf Schwachem Konsistenzmodell
- OpenCL basiert auf RC-Modell
- Bieten Framework, um diese Zugriffe für den Programmierer zu vereinfachen
- OpenCL über Api-Zugriffe, CUDA über C-Entwicklung
- Heute gängige implementierte Form zur Speicherverwaltung

Gliederung

Grundlagen

Untersuchte Speicherkonsistenzmodelle

Gegenüberstellung

Andere Ansätze

Fazit

Fazit

- Es gibt viele Modelle, die unterschiedliche Zwecke erfüllen
- Gut für GPUs: Modelle, die auf Performance und Datenparallelität ausgerichtet sind
- Weak Consistency und Release Consistency Model guter Ansatz
- CUDA und OpenCL bieten Framework und optimierte Modelle
- Bauen auf den vorgestellten Modellen auf
- Je nach Bedürfnis CUDA oder OpenCL verwenden

Vielen Dank für Ihre Aufmerksamkeit!