

Speicherkonsistenzmodelle für GPUs

Marc Blickle



Lehrstuhl für Systemnahe Informatik
und Kommunikationssysteme
Institut für Informatik
Universität Augsburg

2. Februar 2019

- ▶ 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?



Grundlagen

Untersuchte Speicherkonsistenzmodelle

Gegenüberstellung

Andere Ansätze

Fazit



Grundlagen

Untersuchte Speicherkonsistenzmodelle

Gegenüberstellung

Andere Ansätze

Fazit



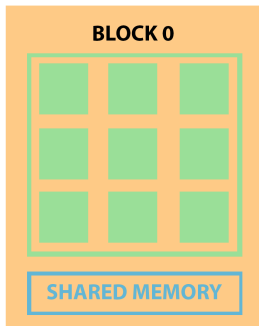
- ▶ GPU-Architektur
- ▶ Speicherkonsistenz
- ▶ Speicherkonsistenzmodelle





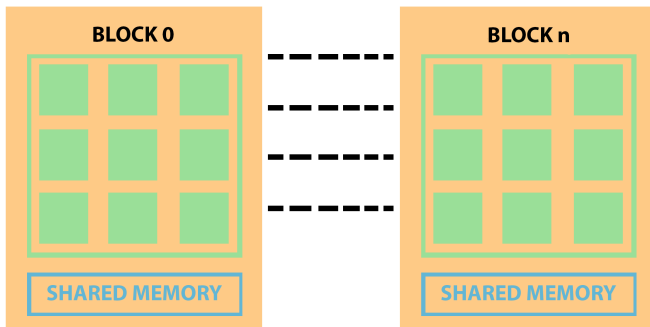
- ▶ Ist eine einzelne Recheneinheit
- ▶ Führt Rechenoperationen aus
- ▶ Interagiert mit Speicher
- ▶ Mehrere tausend Threads

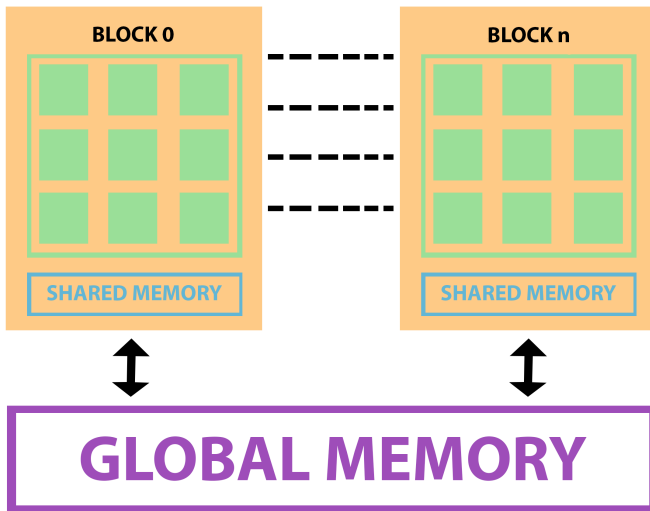




- ▶ Mehrere Threads sind in einem Block zusammengefasst
- ▶ Anzahl unterscheidet sich von Architektur zu Architektur
- ▶ Parallele und oder serielle Ausführung
- ▶ Threads in einem Block teilen Shared Memory
- ▶ Zum Schreiben und Lesen: Über Shared Memory kommunizieren die Threads







- ▶ Mehrere Blöcke bilden ein Gitter
- ▶ Global Memory steht gesamten Gitter zur Verfügung
- ▶ Einzelne Blöcke und somit auch die Threads greifen auf ihn zu und kommunizieren über ihn



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



Speicherkonsistenz und Speicherkonsistenzmodelle



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




```
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;  
    }  
}
```



```
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;  
}
```



- ▶ Speicherdaten im System sollen einheitlich und widerspruchsfrei 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



- ▶ 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?



- ▶ GPU mehr Recheneinheiten als CPU
- ▶ GPU verfügt über tieferen Aufbau
- ▶ Mehr Operationen, mehr parallele Ausführungen

-> GPU braucht Schnelligkeit und Performance, viele Berechnungen

-> Speicher soll geregelt beschrieben und gelesen werden

-> Aber Threads sollen dies schnell und ungehindert tun



Grundlagen

Untersuchte Speicherkonsistenzmodelle

Gegenüberstellung

Andere Ansätze

Fazit



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



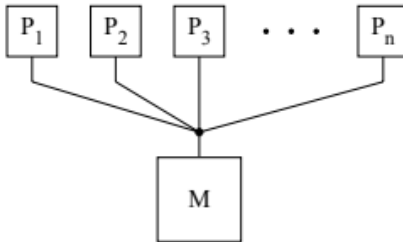
Sequential Consistency Model



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





Program Order

R	R	W	W
R	W	R	W

Konsistenz resultiert aus Programm, wenn:

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

Globale Reihenfolge für alle eingereichten Operationen!



P1:	W(x)1			
P2:	W(y)2			
P3:	R(y)2		R(x)0	R(x)1



- ▶ Ablauf ähnelt stark der Denkweise eines Programmierers
- ▶ 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



- ▶ Zu dem Zweck entwickelt, Speicherlatenz zu verringern
- ▶ Weiterentwicklung aus sequentiell Modell
- ▶ Sequentielle Ordnung wird aufgegeben
- ▶ Leistungssteigerung von 40%
- ▶ Entwickler muss die Speicheroperationen genau anpassen



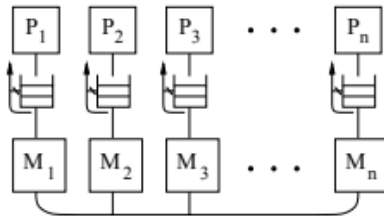
- ▶ Datenoperationen:
 - ▶ Werte werden aus Speicher gelesen oder geschrieben
 - ▶ Andere Prozessoren können nicht immer die bereits errechneten Werte sehen
- ▶ Deshalb Synchronisationsoperationen:
 - ▶ Jeder Prozessor kann diese auslösen
 - ▶ Keine neuen Operationen dürfen dann begonnen und laufende müssen abgeschlossen werden
 - ▶ Synchronisation ist auch Neuordnung von laufenden Operationen
 - ▶ Optimale Reihenfolge zu generieren und Performance gesteigert



Schwache Konsistenz ist gegeben, wenn:

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





Program Order

R	R	W	W
R_s	W_s	R_s	W_s
R_s	R_s	W_s	W_s
R	W	R	W
R	\vdots	\vdots	\vdots
R	W	R	W

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, gefährlich bei kritischen Operationen!
- ▶ Bei zuviel Synchronisation kann Performancegewinn zunichte gemacht werden



Release Consistency Model



- ▶ Weiterentwicklung des schwachen Modells
- ▶ Synchronisationsoperation wird weiter unterteilt

Zweck:

Synchronisationsoperationen sollen nicht mehr warten müssen
Zugriff auf selbe Adresse ist möglich



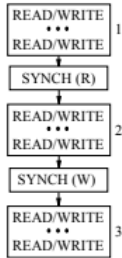
Aquire:

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

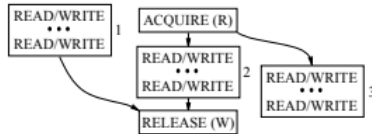
Release:

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





(a) Weak Ordering (WO)

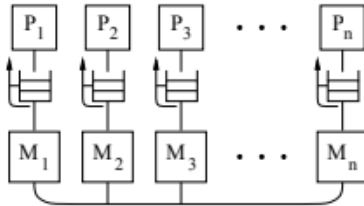


(b) Release Consistency (RC)

Release Consistency ist sichergestellt, wenn:

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





RC

R_c	R_c	W_c	W_c
R_c	W_c	R_c	W_c
R_{acq}	R_{acq}	R	W
R	W	W_{rel}	W_{rel}
R	R	W	W
R	W	R	W

- ▶ Release muss auf vorangegangene Operationen warten
- ▶ Folgende warten aber nicht auf Fertigstellung von Release
- ▶ Acquire muss nicht auf vorangegangene Operationen warten



- ▶ Release muss auf vorangegangene Operationen warten
- ▶ Folgende warten aber nicht auf Fertigstellung von Release
- ▶ Das Ergebnis sind mehr Flexibilität und Schnelligkeit zwischen 'acquire' und 'release'
- ▶ Operationen überlappbar
- ▶ Es gibt insgesamt mehr Operationen
- ▶ Mehr Operationen müssen ausgeführt und geplant werden



Grundlagen

Untersuchte Speicherkonsistenzmodelle

Gegenüberstellung

Andere Ansätze

Fazit



	Vorteile	Nachteile	GPU- geeignet?
Sequentielles Modell	<ul style="list-style-type: none">- Straight forward- Einfach umzusetzen-Kein Synchronisationsaufwand	<ul style="list-style-type: none">- Langsam	Nein
Schwaches Modell	<ul style="list-style-type: none">- Parallele Ausführung von Operationen- Schnell	<ul style="list-style-type: none">- gewisser Synchronisationsaufwand- Entwickler muss genau abstimmen, wann synchronisiert wird	Ja
RC Modell	<ul style="list-style-type: none">- Überlappende Ausführung von Operationen mit gleicher Speicheradresse- Schnell	<ul style="list-style-type: none">- mehr Rechenleistung für gleiche Arbeit nötig- Komplizierter	Ja



Grundlagen

Untersuchte Speicherkonsistenzmodelle

Gegenüberstellung

Andere Ansätze

Fazit



- ▶ 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



Grundlagen

Untersuchte Speicherkonsistenzmodelle

Gegenüberstellung

Andere Ansätze

Fazit



- ▶ Es gibt viele Modelle, die unterschiedliche Zwecke erfüllen
- ▶ Gut für GPUs: Modelle, die auf Performance und Datenparallelität ausgerichtet sind
- ▶ 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!

