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

Wer von Ihnen hat schonmal was von Cryptocurrency gehört?

Wer von Ihnen hat schonmal was von Bitcoin gehört?

Wer hat Bitcoin besessen oder gemined?

Wer weiß, dass Bitcoin mithilfe von Grafikkarten gemined werden?

Nicht der einzige Zweck

Grafikkarten in erster Linie für grafisch Anspruchsvolle Software entwickelt

- 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

linew Seit einem guten Jahrzehnt wird sich vermehrt auf datenparallele Entwicklung bei GPUs konzentriert

Parallele Ausführung beudeutet: Speicherzugriffe auf Shared-Memory müssen vermehrt mithilfe eines sogenannten Speicherkonsistenzmodell abgestimmt werden

Um Fehler zu vermeiden

Bei CPU haben sich Modelle etabliert,

GPU-Hersteller verwenden zwar auch, bieten aber schlechte Transparenz, Doku und Info

Datenparallele Entwicklung wird erschwert

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

Notizen

Vortrag trägt den Titel Speicherkonsistenzmodelle für GPUs

Möchte folgende Fragen klären

Was sind etablierte CPU-Speicherkonsistenzmodelle?

Wie gut oder schlecht sind sie für den GPU-Gebrauch?

Neue Ansätze?

Grundlagen

Untersuchte Speicherkonsistenzmodelle

Gegenüberstellung

Andere Ansätze

Fazit

- GPU-Architektur
- Speicherkonsistenz
- Speicherkonsistenzmodelle

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

Notizen

Ein Thread ist eine einzelne Recheneinheit

Führt Rechenoperationen aus, in denen sie mit Speicher interagiert

Tausende Threads können gleichzeitig arbeiten

- 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

Notizen

Mehrere Threads bilden einen Block

Anzahl Je nach Architektur unterschiedlich

Bis zu 512 Threads pro Block

Parallel oder seriell zueinander

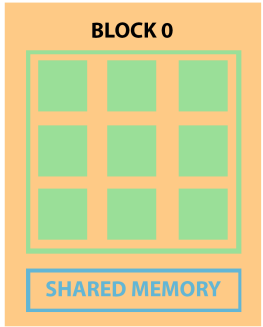
Abhängig von Speichermodell

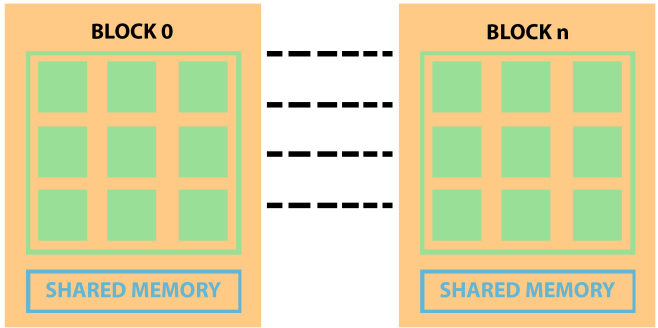
Mehr dazu gleich

Alle Threads im Block teilen sich shared memory

in ihn wird gelesen und geschrieben.

Kommunikation über Shared Memory

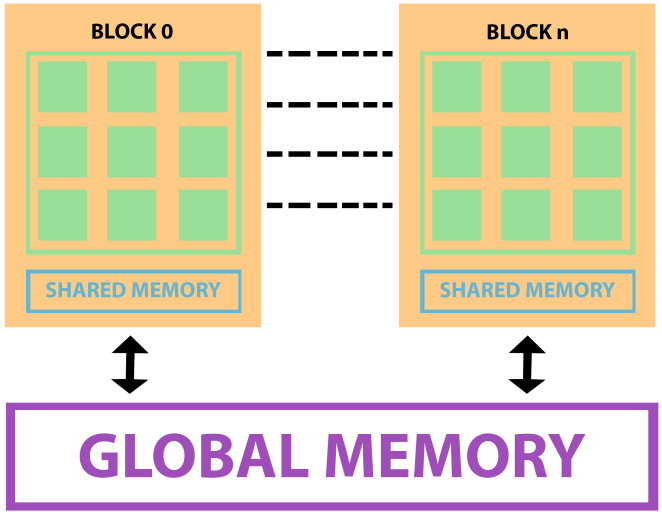

BLOCK 0
SHARED MEMORY



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



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 Speicherzugriffregeln einhalten

Wie Units, Blöcke und Gitter gebildet werden, hängt von der laufenden Software ab

Verschiedene GPUs, verschiedene architekturen, verschiedene Obergrenzen

Bei Softwareprogrammierung zu beachten:

Threads Speicherzugriffsregeln einhalten

es braucht Regelung, wie Threads auf gleichen Speicher zugreifen

ohne sich zu behindern

Es muss Speicherkonsistenz geschaffen werden

Speicherkonsistenz und Speicherkonsistenzmodelle

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

Notizen

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 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

Speicherdaten sollen einheitlich und widerspruchfrei sein

Verhindern, dass Speicherfehler entstehen

Hierzu Anforderungen schaffen, die erfüllt werden müssen

Sie bestimmen, was eine Speicheroperation sehen und machen darf

Verschiedene Architekturen

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

Notizen

Folie 17

Speicherkonsistenzmodell ist Vertrag zwischen Software und Speicher

also Regelwerk, das bestimmt, wie Speicheroperationen ausgeführt werden

Wenn Programmierer und Software es einhalten, kann man mit gewünschten Ergebnissen rechnen

SpeicherKonsistenz ist gewährt

Verschiedene Speicherkonsistenzmodelle

Verschiedene Strukturen

Können für verschiedene Zwecke verwendet werden

Welches Modell ist nun gut für eine GPU?

Notizen

Welches für CPU konzipierte Speichermodell ist jetzt gut für 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

Notizen

GPU sehr viele Recheneinheiten

GPU mit Speicherarchitektur auf mehreren Leveln

Mehr parallele Ausführungen, Threads müssen zusammenarbeiten

GPU muss wegen dieser vielen kleinen Operationen schnell sein

Modell muss auf Performance ausgelegt sein

Speicher soll trotzdem geregelt beschrieben und gelesen werden

Ohne Threads zu blockieren

- 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

Notizen

Abstrahierung

Alle Threads über Switch an SharedMemory gekoppelt

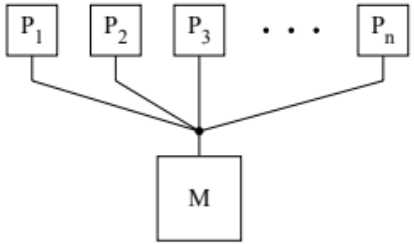
Threads reichen Operationen ein

Switch gibt gesamten Speicher für einen Prozessor frei

Dieser operiert, meldet wenn fertig

Switch sortiert also alle eingehenden Lese und Schreiboperationen und stellt sinnvolle, sequentielle Reihnefolge her

Für alle Prozessoren gilt nun dieselbe Reihenfolge



Program Order

R → R
R → W
W → R
W → W

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!

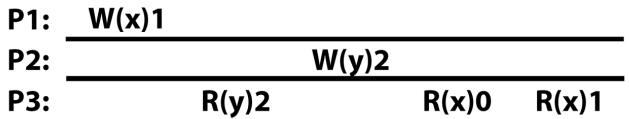
Notizen

Ein Programm, das unter diesem Regelwerk agiert, stellt eine Konsistenz sicher,

Jeder Prozessor Speicheranfragen in der Reihenfolge einreicht, die ihm vom Programm vorgegeben wurde

Speicheranfragen jedes Prozessors von Schlange verarbeitet werden, die die Anfragen sortiert

Für alle eingereichten Operationen wird eine globale Reihenfolge gewählt, die eingehalten werden muss.



Notizen

PRzessor 3 soll Wert erfassen, der nicht existiert

Der wird erst von Prozessor 2 geschrieben

Eigentlich nicht möglich, wegen Compilererror

Aber zeigt Flexibilität

Switch würde nämlich neu anordnen

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

Vorteile:

Ablauf ähnelt stark der Denkweise eines Programmierers

Ist sequentiell und einfach

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

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

Notizen

Zweck: Speicherlatenz verringern (Zeit zischen Anfrage und Erhalt des Wertes vom Prozessor)

Sequentielles Modell so weiterentwickelt, dass zeitweise inkonsistente Modi möglich sind
40% mehr Leistung

Softwareentwickler muss die Speicheroperationen genau anpassen

Wir werden gleich noch sehen wieso

- 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

Notizen

Werte werden aus Speicher gelesen oder geschrieben

Diese Operationen dürfen parallel zueinander ablaufen

Deshalb Synchronisationsoperationen:

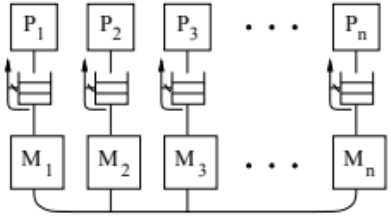
Bringt Shared Memory auf einen Stand

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



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	R	W	W
R	W	R	W

Notizen

Jeder Prozess kann selber auf Memory zugreifen,

Parallel zu den anderen Prozessen

Zwei und Drei Linien: Operation muss abgeschlossen werden, bevor nächste folgen kann

Betrifft ausschließlich Synchronisationsoperationen

Read und Read sind unabhängig voneinander

Gestrichelt: Darf alles parallel ablaufen, außer es betrifft dieselbe Speicheradresse

Führt dazu, dass die Synchronisationsoperationen sequentiell sein müssen, die Datenoperationen aber freier sind

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

(VON FOLIE ABLESEN)

Um diese Bedingungen realisieren zukönnen, ist jede Recheneinheit dazu in der Lage, 2 Arten von Operationen auszuführen

Gewöhnliche Datenoperationen

Synchronisationsoperationen

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

Vorteile

Einzelne Threads können sich in beliebiger Reihenfolge Zugriff verschaffen

Müssen nicht auf andere Prozesse warten

Speicherlatenz wird verringert, Performance gesteigert bis 60%

Nachteile

Aufwändigere Programmierung, umfangreichere Regeln

Fehleranfälliger

Bei zuviel Synchronisation kann Performancegewinn zunichte gemacht werden

Release Consistency Model

- Weiterentwicklung des schwachen Modells
- Sychronisationsoperation wird weiter unterteilt

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

Weiterentwicklung des schwachen Modells

Sychronisationsoperation wird weiter unterteilt

Zweck: Synchronisationsoperationen sollen nicht mehr warten müssen

Read auf selbe Adresse immer 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

Aquire

Überprüft nur, ob alle Schreiboperationen auf den gemeinsamen Dateien fertig

Falls ja, erhält der 'aquire'-ausführende Prozessor Zugriff

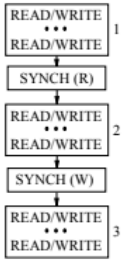
Funktion zum Anfragen von Berechtigungen

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 '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

Notizen

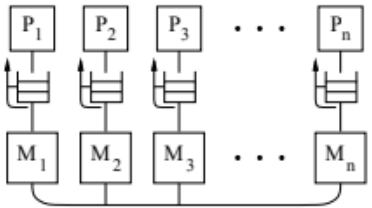
Release Consistency ist sicher gestellt wenn

Alle Schreiboperationen von A von B gesehen werden, nachdem A sie released hat und bevor B sie aquired hat

Alle 'aquire'-Operationen durchlaufen sind, bevor geschrieben oder gelesen wird

Alle Schreib- oder Leseoperationen durchlaufen sind, bevor 'release' startet

Die nächste Abbildung verdeutlicht das



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 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

Notizen

Folie 40

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

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

Vorteile

Das Ergebnis sind mehr Flexibilität und Schnelligkeit zwischen 'acquire' und 'release'

Operationen überlappbar

Nachteile

es gibt insgesamt mehr Operationen

Mehr Operationen müssen ausgeführt und geplant werden

	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

- 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

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

- 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

Es gibt viele Modelle, die unterschiedliche Zwecke erfüllen

Gut für GPUs: Modelle, die auf Performance und Datenparallelität ausgerichtet sind

Es ist zu beachten, dass es CPU MOdelle sind, deshalb 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!