Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Dennis Luxen, Vitaly Osipov,
Dennis Schieferdecker

Algorithm Engineering - Graphikprozessoren SS 2010
Übungsblatt 4

http://algo2.iti.kit.edu/cuda_10.php
{luxen|osipov|schieferdecker}@kit.edu
Ausgabe: Dienstag, 11.05.2010
Abgabe: Montag, 17.05.2010

Korrektur 14.05.2010: Definition von Δ in Aufgabe 2b) geändert.

Aufgabe 1 (*Parallele Reduktion II*)

Ausgehend von Übungsblatt 3 werden wir weitere Möglichkeiten vorstellen, wie die Ausführungszeit der parallelen Reduktion weiter gesenkt werden kann. Sie können entweder Ihren eigenen Kernel weiterverwenden oder den von uns zur Verfügung gestellten Kernel anpassen.
Testen Sie die Laufzeit Ihres Reduktionsverfahrens wie auf dem letzten Übungsblatt. Um Fehler leichter zu finden, könnte es sinnvoll sein, zunächst wieder nur auf einem Block parallel zu arbeiten.

a) Die Hälfte aller Threads in einem Block wird nur einmal benötigt, um die Daten vom globalen Speicher in den *shared memory* zu laden. Man kann diesen Ladevorgang mit dem ersten Schritt kombinieren und nur direkt die Summen benachbarter Elemente in den *shared memory* laden. Dies verringert den Anteil inaktiver Threads für einen *kernel* Aufruf beträchtlich. Nun kann die benötigte Anzahl Threads pro Block halbiert werden. Auf diese Weise summiert jeder Block weiterhin über die gleiche Anzahl Elemente, braucht aber nur noch halb so viel *shared memory*. Alternativ lässt man die Anzahl Threads konstant und benötigt insgesamt weniger Blöcke. Untersuchen Sie beide Varianten.

b) Die letzten 5 Schritte in einem *kernel* Aufruf verwenden weniger als 32 Threads. Um dies zu vermeiden, können diese Schritte *ausgerollt* werden. Dies bedeutet, anstatt eine `for` Schleife zu verwenden, die entsprechenden Befehle direkt hintereinander zu schreiben. Eine Synchronisierung muss dabei nicht erfolgen, wenn Sie die ausgerollten Befehle in eine eigene Funktion auslagern und den Zeiger auf das *shared memory* als `volatile` übergeben. Dies stellt sicher, dass Befehle wie z.B. `shared_memory[tid] += shared_memory[tid+i]` wirklich nur auf dem *shared memory* arbeiten. Ansonsten kann es passieren, dass der Compiler Werte in Registern zwischenspeichert, die noch nicht wieder zurückgeschrieben sind, wenn ein anderer Thread darauf zugreift.

c) *(Zusatz–freiwillig)* Die in Teilaufgabe a) eingeführte Maßnahme kann erweitert werden, indem nicht nur der erste Schritt des Reduktionsverfahrens implizit beim Laden ausgeführt wird, sondern bereits die ersten k Schritte. Halten Sie die Anzahl Threads pro Block konstant und untersuchen Sie Auswirkung verschiedener Werte von k auf die Laufzeit. *Hinweis:* Das Zusammenfassen der ersten k Schritte kann geschickter gelöst werden, anstatt den parallelen Algorithmus seriell in einem Thread auszuführen.

Aufgabe 2 (*Präfixsummen*)

Präfixsummen (genauer Präfixsummenarrays) sind ein wichtiger Baustein in vielen (parallelen) Algorithmen. Glücklicherweise kann ihre Berechnung sehr effizient parallelisiert werden. Ein Präfixsummenarray ist wie folgt definiert: Gegeben sei ein assoziativer Operator $\bigoplus$ sowie ein Array a der Länge n. Das zugehörige Präfixsummenarray p ist ein Array gleicher Länge mit der Eigenschaft $p[k] = \bigoplus_{i=1}^{k} a[i],\ 0 < k \leq n$. Jedes Element p enthält also die Summe bzgl. $\bigoplus$ der Elemente aus a bis einschließlich zum gleichen Index. Diese Definition wird auch als *inklusiv* bezeichnet, da die Summe bis *einschließlich* des gleichen Indexes genommen wird. Entsprechend bedeutet *exklusiv*, dass die Summe bis *ausschließlich* des gleichen Indexes gebildet wird.
Für die folgenden Aufgaben können Sie wieder davon ausgehen, dass n eine Zweierpotenz ist. Verwenden Sie auch wieder einen Array mit $n = 2^{24}$ zufälligen Ganzzahlen aus $\{0, \ldots, 10\}$.

a) Betrachten Sie zunächst nur die Berechnung von Präfixsummen innerhalb eines Blockes. Wählen Sie dafür eine geeignete Blockgröße b. Ein einfacher paralleler Algorithmus arbeitet in $\lceil \log b \rceil$ Schritten. Nach Schritt m ist die Invariante $a[k] = \bigoplus_{i=\max(1,k-2^m+1)}^{k} a_{start}[i]$, $0 < k \leq b$ erfüllt. Hierbei bezeichnet a_{start} die Werte von a zu Beginn des Algorithmus'. Vergewissern Sie sich, dass sich nach Schritt m die ersten 2^m Arrayeinträge nicht mehr ändern und das korrekte Ergebnis enthalten. Geben Sie die benötigte Anzahl Additionen in Abhängigkeit von b im O-Kalkül an. Nehmen Sie an, dass ein Block beliebig groß werden kann. Wieviele Additionen würde eine naive serielle Implementierung benötigen? Messen Sie anschließend die Laufzeit für Ihren Algorithmus.

b) Eine effizientere Variante der Präfixsummenberechnung läuft in 2 Phasen ab. Die erste Phase entspricht einer parallelen Reduktion. Nach dieser Phase gilt $a[k] = \bigoplus_{i=k-\Delta}^{k} a_{start}[i]$, $\Delta = max\{2^x - \mathbf{1} | k = 0 \pmod{2^x}, x \geq 0\},\ 0 < k \leq b$. Vergewissern Sie sich, dass dieses Ergebnis dem Ergebnis des Algorithmus' aus Übungsblatt 3 Aufgabe 2a) in umgekehrter Reihenfolge entspricht. Die zweite Phase läuft invers zu Phase 1 (also von wenigen zu vielen aktiven Threads). Sie sammelt die durch den Reduktionsalgorithmus berechneten Zwischenergebnisse ein und addiert sie geschickt auf die anderen Zellen. Da die Variante für *exklusive* Präfixsummen einfacher ist, wenden wir folgenden Trick an. Wir kopieren $a[b]$ auf $a[b+1]$ und setzen $a[b] = 0$. Die zweite Phase wird weiterhin auf $a[1], \ldots, a[b]$ ausgeführt. Nach Beendigung von Phase 2 wird $a[2], \ldots, a[b+1]$ die gewünschte Lösung enthalten. Die zweite Phase hat $\lceil \log b \rceil$ Schritte. In Schritt m werden werden die Elemente in $a[i]$ auf die Positionen $a[i+\Delta]$ addiert und anschließend die alten Werte von $a[i+\Delta]$ an die Positionen $a[i]$ kopiert, mit $i \in \{x | x = 0 \pmod{2^{-m+\lceil \log b \rceil}}\}, \Delta = 2^{-m+\lceil \log b \rceil}$. Wieviele Additionen und Vertauschungen in Abhängigkeit von n benötigt diese Variante? Wie lange benötigt Ihr Algorithmus für die Ausführung?

Hinweis: Zur Verdeutlichung der Funktionsweise des Verfahrens ist es hilfreich, sich den Ablauf als Binärbaum vorzustellen mit Positionen i und $i+\Delta$ in Schritt m als Kinder von Position $i+\Delta$ in Schritt $m-1$.

c) *(Zusatz–freiwillig)* Übertragen Sie beide Maßnahmen zur Beschleunigung des Reduktionsalgorithmus' des letzten Übungsblatts auf Ihren Präfixsummenalgorithmus, um diesen zu verbessern. Geben Sie die neue Laufzeit Ihres Algorithmus' an.

d) Erweitern Sie die Berechnung der Präfixsummen für den gesamten Array. Gehen Sie entsprechend der Parallelisierung des Reduktionsverfahrens vor und verwenden Sie Ihren Algorithmus aus Teilaufgabe a) als Baustein. Sie benötigen zusätzlich einen *kernel*, der einen konstanten Wert auf einen Bereich des Arrays addiert. Testen Sie Ihren Algorithmus mit den Varianten aus Teilaufgabe a) und b).

Allgemeine Hinweise:

- Innerhalb einer *kernel* Funktion können *shared memory* Arrays nur statisch alloziert werden. Vom *host* aus ist auch eine dynamische Allokation möglich. Hierbei erfolgt die Variablendeklaration im Kernel durch `extern __shared__ float var[];`. Der Kernelaufruf `myKernel<<<bl, th, dyn>>>(params);` übergibt im dritten Parameter *dyn* die Größe in Byte, die für dynamische *shared memory* Variablen pro Block reserviert werden soll. Beachten Sie, dass alle auf diese Weise deklarierten Variablen in einem Block an der selben Adresse starten.

- Variablen im *shared memory* können auch außerhalb von `__device__` oder `__global__` Funktionen deklariert werden. In diesem Fall erstreckt sich die Sichtbarkeit der Variablen über alle Funktionen, die auf der Graphikkarte ablaufen.

- Ein *kernel*-Aufruf erfolgt asynchron, d.h. der Hostcode wird sofort weiterausgeführt ohne auf das Ende der GPU-Berechnung zu warten. Für korrekte Laufzeitmessungen muss deshalb vor der Timerabfrage `cudaThreadSynchronize();` aufgerufen werden. So wird sicher gestellt, dass alle auf der GPU laufenden Threads abgearbeitet sind, bevor die Zeit ausgelesen wird. Vor einem *kernel* Aufruf oder einer `cudaMemcpy` Anweisung findet außerdem eine implizite Synchronisierung statt.

Lernziele – Woche 4

Nach Bearbeitung des vorliegenden Übungsblattes sollten Sie mit folgenden Themen vertraut sein:

- *Beschleunigungstricks*

- *Parallele Scanalgorithmen*

- Mehr über *shared memory* Variablen