Karlsruher Institut für Technologie
Institut für Theoretische Informatik

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

Algorithm Engineering - Graphikprozessoren SS 2010
Übungsblatt 3

http://algo2.iti.kit.edu/cuda_10.php
{luxen|osipov|schieferdecker}@kit.edu
Ausgabe: Dienstag, 04.05.2010
Abgabe: Montag, 10.05.2010

Hinweis: Bei Problemen oder Fragen stehen wir jederzeit zur Verfügung

Ab sofort steht Ihnen der Rechenknoten i10pc112 zur Verfügung. Er ist ausgestattet mit einer Tesla C870 sowie einer GeForce 8400 GS. Verwenden Sie bitte in Zukunft nur noch diesen Rechner. Beachten Sie den kleineren Speicherausbau der beiden Graphikkarten.
Machen Sie bitte auch Gebrauch von unserem *Compute Server Scheduler*, damit sich verschiedene Anwendungen nicht gegenseitig behindern. Für Sie bedeutet dass, `'exclusive '` vor jeden Programmaufruf zu stellen. Das System wird jeden Prozess mit diesem Aufruf einzeln ausführen. Programme, die ohne `exclusive` ausgeführt werden, werden unter Umständen abgebrochen.
Außerdem möchten wir Sie auf die, auf der Homepage verlinkte, Mailingliste hinweisen, die wir eingerichtet haben. Sie dient für kurzfristige Bekanntmachungen, aber auch zur Diskussion unter den Teilnehmern und mit den Betreuern.

Aufgabe 1 (*Theorie*)

Die Recheneinheiten (SP - *streaming processor*) aktueller GPGPU-Karten von Nvidia sind in Gruppen zu je 8 Einheiten zusammengefasst (SM - *streaming multiprocessor*). Ein SM kann bis zu 8 Blöcke mit zusammen maximal 768 (G80) bzw. 1024 (G200) Threads verwalten. Desweiteren stehen jedem SM 16 kByte *shared memory* und 8192 (G80) bzw. 16384 (G200) Register (nicht Byte!) für 32bit Werte zur Verfügung. Während der Ausführung werden jedem SM nur so viele Blöcke zugeteilt, so dass keine dieser Beschränkungen verletzt wird.
Innerhalb des SM sind Blöcke in *warps* zu je 32 Threads organisiert. Besitzt ein Block eine nicht durch 32 teilbare Anzahl Threads, wird der letzte *warp* mit Dummythreads gefüllt. Der Scheduler eines SM teilt immer genau einen *warp* zur Ausführung ein. Die Ausführung eines Befehls für einen *warp* dauert 4 Taktzyklen, da jeder SM nur 8 Recheneinheiten besitzt. Um Wartezeiten zu verbergen (*latency hiding*), wie sie z.B. beim Zugriff auf den globalen Speicher auftreten, kann der SM sehr schnell zwischen einzelnen *warps* umschalten (*zero-overhead thread scheduling*). Damit möglichst wenig echte Wartezeiten auftreten, sollte ein SM die maximale Anzahl *warps* ausreizen, so dass immer ein nicht wartender *warp* gefunden wird.

a) Wieviele *warps* kann ein SM maximal verwalten?

b) Ein Kernel verwendet 17 lokale 32bit Variablen und 2 kByte *shared memory*. Nehmen Sie eine Gridgröße von $(128, 128, 1)$ und eine Blockgröße von $(16, 16, 1)$ an. Aus wievielen *warps* kann der Scheduler einen SM wählen? Wieviele Threads werden parallel ausgeführt? (Hinweis: lokale Variablen werden in Registern gespeichert)

c) Was ist eine optimale Blockgröße für obiges Beispiel? In diesem Zusammenhang bedeutet optimal, dass die zur Verfügung stehenden Ressourcen möglichst komplett ausgenutzt werden. Verwenden Sie eine zweidimensionale Blockorganisation.

Gehen Sie bei den Aufgaben von einer G80-Architektur wie auf der Tesla C870 und von einer G200-Architektur wie auf der Tesla C1060 aus.

Aufgabe 2 (*Parallele Reduktion*)

Ein Reduktionsalgorithmus dient zur Berechnung von $\bigoplus_{i=1}^{n} a[i]$ auf einer Liste a der Länge n, wobei $\bigoplus$ eine assoziative Operation darstellt. Die Berechnung der Summe über alle Listenelemente stellt z.B. einen Reduktionsalgorithmus dar.
Im Folgenden nehmen wir der Einfachheit halber an, n sei eine Zweierpotenz. Außerdem definieren wir $\bigoplus$ als normale Addition auf den natürlichen Zahlen. Ein einfaches Verfahren zur parallelen Reduktion arbeitet wie folgt in $\log n$ Schritten: In Schritt k werden die Elemente an Position $i = 2^k \pmod{2^{k+1}}$ auf die Elemente an Position $i = 0 \pmod{2^{k+1}}$ addiert. Dieses Verfahren arbeitet *in place*, d.h. Elemente der Liste werden bei der Ausführung durch Teilsummen bzw. die Endsumme ersetzt.
In den folgenden Teilaufgaben sind Sie aufgefordert, das Basisverfahren sowie einige Verbesserungen zu implementieren. Ermitteln Sie für jede dieser Varianten die Laufzeit und geben Sie die berechnete Summe an. Verwenden Sie eine Liste aus 2^{24} zufälligen Ganzzahlen aus $\{0, \ldots, 10\}$.

a) Implementieren Sie das einfache parallele Verfahren zunächst nur für einen Block. Die Ergebnisse der einzelnen Blöcke können Sie anschließend seriell zusammenzählen. Verfahren Sie wie folgt: Jeder Thread ist für genau ein Listenelement zuständig (d.h. bei einer Blockgröße von 256 können Sie die Summe von 256 Elementen berechnen). Laden Sie die Elemente der Liste in das *shared memory*. Arbeiten Sie anschließend die Schritte im *shared memory* ab. Denken Sie daran, an den passenden Stellen eine Threadsynchronisation durchzuführen. Am Ende des Kernel muss nur das Ergebnis zurück in den globalen Speicher geschrieben werden.

b) Die Ausführungszeit des Verfahrens kann durch zwei Maßnahmen stark beschleunigt werden. (1) Stellen Sie sicher, dass in möglichst Schritten alle Threads eines *warps* dieselben Befehle ausführen. Entweder soll jeder der jeweils 32 Threads addieren oder nichts tun, aber keine Mischform.

   Hintergrund: Gibt es im Programm eine Verzweigungsmöglichkeit und es folgen nicht alle Threads eines warps *dem gleichen Ausführungspfad, so wird der* warp *für den verzweigten Abschnitt zweimal ausgeführt. Dabei warten die Thredas, die nicht zur aktuellen Verzweigungsmöglichkeit gehören.*

   (2) Stellen Sie sicher, dass die Threads auf hintereinander liegende Speicherbereiche im *shared memory* zugreifen. Dafür kann es nötig sein, die oben angegebene Berechnungsregel zu modifizieren.

   Hintergrund: Der shared memory *ist in* 16 *Bänken zu je* 1 *KByte Speicher organisiert. Aufeinander folgende* 32*bit Werte werden in aufeinander folgenden Bänken gespeichert. Zugriffe auf unterschiedliche Bänke finden parallel statt. Bei gleichzeitigem Zugriff auf unterschiedliche Elemente einer Bank muss ein Thread warten. Ein Zugriff dauert* 2 *Takte.*

c) Erweitern Sie Ihr Verfahren, so dass auch die Ergebnisse der Blöcke wieder parallel addiert werden. Sie können bei geschickter Wahl der Indizes den gleichen Algorithmus wie für die Addition innerhalb eines Blockes verwenden.

Lernziele – Woche 3

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

- *Threadverwaltung* in CUDA
- *Parallele Reduktion*