# Übungsblatt 4

18.05.2018 / B. Leder

Wissenschaftliches Rechnen III / CP III

## Aufgabe 4.1: *Parallele Reduktion*

Eine häufige Aufgabe im parallelen Rechnen ist die Reduktion, z.B. die Summe aller Elemente eines Vektor/Array (es sind andere assoziative und kommutative Operationen denkbar: Produkt, Maximum, Minimum, ...)

```
int sum = 0;
for (int i=0; i<n; i++)
   sum += array[i];
```

Dies kann parallel berechnet werden:

1) Einteilung des gesamten Array in kleinere Teilmengen
2) Je ein Thread berechnet die Summe für ein Teilmenge
3) Teilsummen werden aufsummiert

Der letzte Schritt kann entweder seriell geschehen oder durch Iteration von 1)-2) mit dem reduzierten Array der Teilsummen. Ein verbreiteter Ansatz ist in 1) Teilmengen der Größe 2, also Paare zu wählen. Da die Anzahl der Summanden sich in jeder Iteration halbiert, ist die Summe in $\log_2(n)$ Iterationen berechnet (dafür werden genau $n-1$ Additionen benötigt, wie im seriellen Fall). In jedem Schritt werden aber immer weniger Threads gebraucht.

In der Datei `reduction.cu` finden Sie eine Implementierungen der Summe eines Array von Integern:

a) auf der CPU als Rekursion (statt Iteration) mit `for`-Schleife über Paare, die immer genau $n/2$ voneinander enfernt sind (`recursiveReduce`)
b) auf der GPU als parallel Reduktion pro Thread-Block mit $n_b$ Threads, d.h. $\log_2(n_b)$ Iterationen mit Teilmengen der Größe 2 (`reduceInterleaved`) und anschliessender serieller Summation der Blocksummen auf dem Host
c) eine verbesserte Variante von b) wobei *jeder* Thread als erstes zwei Elemente summiert und dann die paarweise Reduktion pro Thread-Block beginnt (`reduceUnrolling`)

***Hinweise:***

- die execution configuration ist 1-dimesnsional (1D-Grid, 1D-Blöcke)
- in Version c) werden halb so viele Blöcke benötigt wie in Version b)
- `__syncthreads();` erzwingt Synchronisation der Threads im Thread-Block
- laden Sie auch die Datei `common.h` herunter und speichern sie im selben Verzeichnis wie `reduction.cu`, sie ist ein Beispiel für eine gemeinsame Header-Datei, in der globale Variablen, Makros und Funktionen definiert werden können
- kompilieren Sie mit `nvcc -O3 -arch sm_30 reduction.cu`

***Aufgaben:***

i) Skizzieren Sie für c) den Ablauf der Reduktion eines Arrays mit $n = 16$ Elementen und einer execution configuration mit 4x1 Blöcken und 2x1 Grid. Kennzeichnen Sie dabei die Aufteilung der Daten auf die Thread-Blöcke und die Synchronisation.

ii) Messen Sie die Laufzeiten für $n = 2^{24}$ (`int n = 1 << 24;`) und verschiedene Blockgrößen (Vielfache von 32) mit

```
nvprof ./a.out
```

iii) Messen Sie auch die Werte für "Global Memory Load Efficiency" ($< 100\%$ wenn Daten unnötig mehrmals gelesen werden müssen), "Global Load Throughput" (erreichte Bandbreite) und "Achieved Occupancy" (Auslastung) mit

```
nvprof --metrics gld_efficiency,gld_throughput,achieved_occupancy  ./a.out
```

iv) Versuchen Sie mit den gemessenen Größen die Laufzeitunterschiede zu erklären.

v) Implementieren Sie eine Version, die am Anfang $q > 2$ Elemente in *jedem* Thread summiert bevor die Reduktion pro Thread-Block beginnt. Bestimment Sie das Optimum von $q$ für $n = 2^{24}$.

vi) Implementieren Sie ausgehend von Kernel b), oder Ihrer optimierten Version, eine Reduktion für den Datentyp `double`. Gibt es Unterschiede in der Laufzeit und den in ii) gemessenen Größen?

*20 Punkte*