Prüfungsaufgabe 3

In dieser Aufgabe soll das Sortieren eines Feldes von `cl_uint`-Werten mit Hilfe des **Radix-Sort**-Algorithmus in OpenCL implementiert werden. Dies soll in 4 Schritten geschehen, in denen das Feld – beginnend beim niedrigwertigsten Byte - jeweils nach einem Byte sortiert wird.

Für das gerade betrachtete Byte wird zunächst gezählt, wie häufig die möglichen Werte dieses Bytes (die bekanntermaßen zwischen 0 und 255 liegen) im Feld vorkommen. Die Ergebnisse dieses Zählvorgangs sollen in ein Feld aus 256 `cl_uint`-Elementen eingetragen werden.

Auf der CPU könnte das z.B. folgendermaßen implementiert werden:

```
void countValues(int feld[], int size, int byteNr, int counts[]){
   int shiftbits = 8*byteNr;
   for (int i=0 ; i<256 ; i++)
      counts[i]=0;
   for (int i=0 ; i<size ; i++) {
      unsigned char val = (feld[i]>>shiftbits)&255;
      counts[val]++;
   }
}
...
```

Anschließend werden für das Feld `counts` die **Präfixsummen** berechnet. Das Feld mit den Präfixsummen enthält dann für jeden möglichen Wert des betrachteten Bytes den Startindex, bei dem in einem nach dem betrachteten Byte sortierten Feld, die Elemente mit dem entsprechenden Wert beginnen.

Nun können die Feldelemente mit Hilfe der Präfixsummen nach dem betrachteten Byte sortiert in ein neues Feld eingetragen werden.

Auf der CPU könnte das z.B. folgendermaßen implementiert werden:

```
void insertSortet(cl_int source[], cl_int dest[], int size,
                  cl_int praefix[], int byteNr){

   int shiftbits = 8*byteNr;
   for (int i=0 ; i<size ; i++) {
      unsigned char val = (source[i]>>shiftbits)&255;
      dest[praefix[val]++]= source[i];
   }
}
```

Der gesamte Sortier-Algorithmus kann dann auf der CPU z.B. so formuliert werden:

```
void radixsort(int feld[], int size)
{
   int *feld2 = new int[size];     // temporary array
   int counts[256];
   uint praefix[256];
   int byteNr;                // 0,1,2 or 4 (0 for LSB)

   for (byteNr=0 ; byteNr<4 ; byteNr ++) {
      countValues(feld, size, byteNr, counts);
```

Prüfungsaufgabe 3

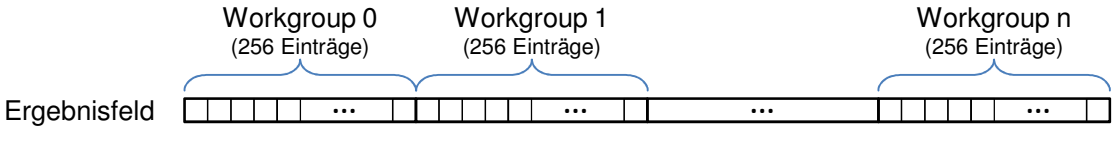
```
            calcPraefix(counts, praefix, 256);
            if (byteNr%2==0)
               insertSortet(feld, feld2, size, praefix, byteNr);
            else
               insertSortet(feld2, feld, size, praefix, byteNr);
         }
         delete [] feld2;
}
```

Aufgabe:

Implementieren Sie diesen Algorithmus in OpenCL.

Gehen Sie für das Zählen der Häufigkeit der einzelnen Byte-Werte wie folgt vor:

- Um eine sinnvolle Parallelisierung zu erreichen, muss die Aufgabe auf eine größere Zahl von Workitems aufgeteilt werden, die alle dann ein oder mehrere Feldelemente bearbeiten.
- Implementieren Sie dazu einen Kernel **`calcStatistic`**, der für jedes Feldelement den Wert des betrachteten Bytes bestimmt und die Anzahl der Feldelemente zu jedem Bytewert (von 0 bis 255) für jede Workgroup separat in einem Feld vom Typ `int` im globalen Memory speichert (siehe Skizze 1):



Skizze 1: Ergebnisfeld des Kernels `calcStatistic`, das die Häufigkeit der Feldelemente zu den 256 möglichen Bytewerten für jede Workgroup separat enthält

  - Der Kernel benötigt als Eingabeparameter das Feld, dessen Länge und ein weiteres Feld vom Typ `int` der Länge `nr_workgroups*256` für die Ergebnisse der einzelnen Workgroups. (`nr_workgroups` ist dabei die Anzahl der Workgroups)
  - Bearbeiten Sie in jedem Workitem (in einer Schleife) 256 Feldelemente. Benutzen Sie eine Workgroup-Größe von 32. (**Achtung**: bitte Hinweis auf Seite 4 beachten)
    Jede Workgroup bearbeitet damit 32*256= 8192 Feldelemente. Die globale Worksize erhält man dann, wenn man die Feldgröße auf das nächste Vielfache von 8192 vergrößert und diese Zahl durch 256 dividiert:

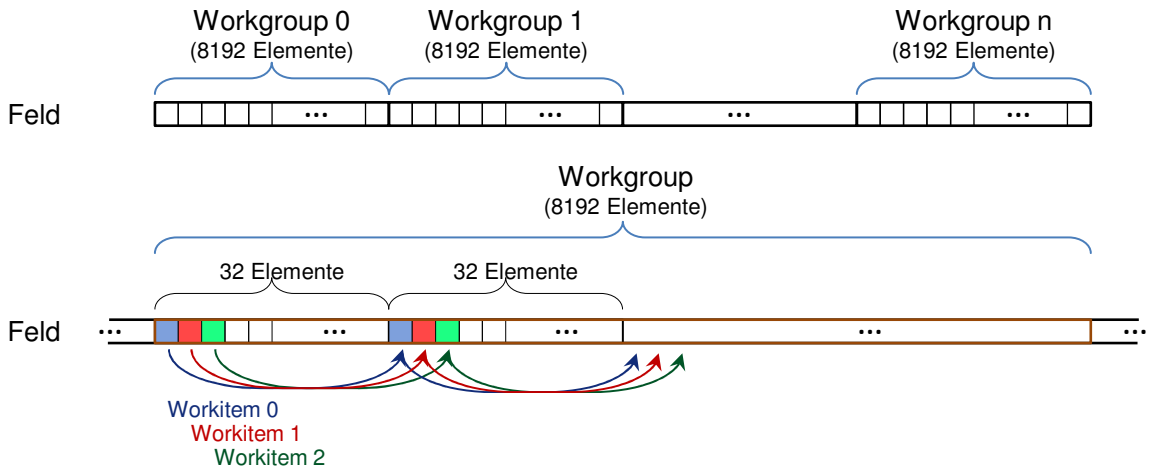
    ```
    global_worksize = (size+8191)/8192*8192/256
                    = (size+8191)/8192*32
    ```

    (Das Feld selber muss nicht vergrößert werden. Sie müssen aber in den Kernels sicherstellen, dass keine Zugriffe über das Feldende hinaus stattfinden.) Die Anzahl der Workgroups ist dann:

    ```
    nr_workgroups = global_worksize/32
    ```

  - Damit die Speicherzugriffe auf das Feld effizient erfolgen können, soll jedes Workitem innerhalb des Element-Blocks, den die Workgroup bearbeitet, beim LX-ten Element beginnen und dann in einer Schleife von da an jedes 32-te Element bearbeiten. (LX ist dabei die local_id(0)) (Siehe Skizze 2)

Prüfungsaufgabe 3



Skizze 2: Aufteilung der Feldelemente auf Workgroups und Workitems

- Würden mehrere Workitems bei der Berechnung der Statistik die Elementzahlen im selben Feld aufsummieren, könnte nicht ausgeschlossen werden, dass mehrere Workitems gleichzeitig versuchen, denselben Eintrag zu inkrementieren. Dies würde zu einem falschen Ergebnis führen. Deshalb benötigt jedes Workitem temporär ein eigenes Feld zum Aufsummieren der Element-Anzahlen. Benutzen Sie dazu ein zweidimensionales Feld im lokalen Memory

  ```
  local int counts[32][256];
  ```

  Jedes Workitem hat so exklusiv ein Feld mit 256 Elementen zur Verfügung, wenn als erster Index die local_id verwendet wird.

  Anstatt für jedes Workitem ein exklusives Feld zum Aufsummieren der Zahlen zu verwenden, wäre es auch möglich, mit Atomic-Operations zu arbeiten. Die Aufgabenstellung fordert aber explizit die Lösung mit den exklusiven Feldern.

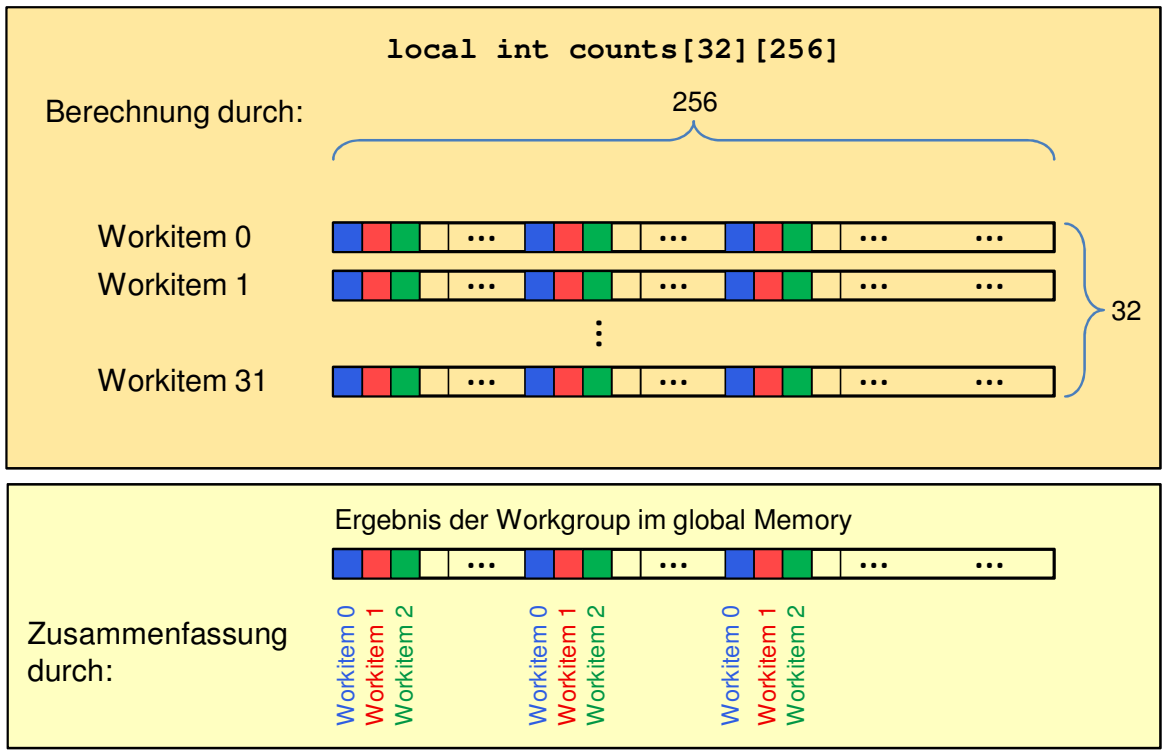
- Wenn alle 32 Workitems einer Workgroup ihre Elemente bearbeitet haben, müssen die dabei entstandenen 32 Felder zum Gesamtergebnis der Workgroup zusammengefasst und ins globale Memory geschrieben werden. Um effiziente Memory-Zugriffe (sowohl im lokalen als auch im globalem Memory) zu gewährleisten, soll dazu jedes Workitem LX die folgenden Summen berechnen:

$$\sum_{k=0}^{31} counts[k][LX + i*32] \quad \text{für} \quad i = 0,...,7$$

  (siehe dazu Skizze 3)

- Implementieren Sie einen weiteren Kernel **`reduceStatistic`**, der die Ergebnisse der einzelnen Workgroups addiert und so die Gesamtstatistik für das Feld berechnet. Verwenden Sie dazu eine Workgroup mit 256 Workitems. Addieren Sie in jedem Workitem die Zahlen zu einem Byte-Wert. Der Kernel benötigt als Eingabeparameter das Ergebnisfeld des vorigen Kernels **`calcStatistic`** (Skizze 1) sowie die Anzahl der dabei verwendeten Workgroups. Der Kernel **`reduceStatistic`** benötigt für das Endergebnis **kein** weiteres Feld. Die ersten 256 Einträge des Eingabefeldes können durch das Endergebnis überschrieben werden.

Prüfungsaufgabe 3



Skizze 3: Aufteilung der Berechnung und der Zusammenfassung auf die Workitems einer Workgroup

Berechnung der Präfixsummen:

- Verwenden Sie eine Präfixsummen-Lösung in OpenCL. Da die Präfixsummen nur für ein Feld der Länge 256 berechnet werden müssen, ist dies relativ einfach und wie in der Vorlesung besprochen umsetzbar.

Gehen Sie für das sortierte Eintragen der Feldelemente in das andere Feld wie folgt vor:

- Verwenden Sie eine Workgroup mit 256 Workitems.
- Kopieren Sie das Feld mit den Präfixsummen ins lokale Memory
- Um effiziente Memory-Zugriffe (sowohl im lokalen als auch im globalem Memory) zu gewährleisten, soll jedes Workitem LX in einer Schleife die Feldelemente mit den Indizes LX+256*k (k ist dabei die Schleifenvariable) bearbeiten.

  (analog zur unteren Darstellung in Skizze 2, nur mit 256 Workitems)
- Verwenden Sie Atomic-Operations um die Indizes im Präfixsummenfeld kollisionsfrei zu inkrementieren.

Hinweise:

- Vor dem Aufruf des ersten Kernels muss das Eingabe-Feld ins Grafikkarten-Memory übertragen werden. Zwischenergebnisse müssen NICHT ins main-Memory zurück übertragen werden. Erst am Ende muss das sortierte Feld ins main-Memory übertragen werden.
- **Achtung:** Der Kernel **`calcStatistic`** benötigt für das Feld `counts` 32 KByte lokales Memory pro Compute Unit, was für aktuelle Grafikkarten kein Problem ist. Alte Karten besitzen aber teilweise nur weniger. Die Workgroup-Größe muss dann

entsprechend reduziert werden. (z.B. besitzen die Gforce-Karten vonVidia bis Gforce 295 nur 16KByte lokales Memory pro Compute Unit, wovon der Compiler auch noch ein paar Bytes für andere Dinge benutzt. Die Workgroup-Größe muss hier deshalb auf 8 Workitems reduziert werden.)

- Wenn die 4 Bytes eines Feldelementes als 32-Bit `int`-Wert angesprochen werden, hängt es von der *Endianness* des Prozessors und der GPU ab, in welcher Reichenfolge die 4 Werte vorliegen. Wenn Sie wie im o.a. Beispiel für den CPU-Code mit Shift-Operatoren arbeiten, spielt dies aber keine Rolle. Wenn Sie die `int`-Werte in `uchar4`-Werte umwandeln würden, müssten Sie die Bytes aber je nach *Endianness* des Prozessors und der GPU in anderer Reihenfolge bearbeiten.

Bonusaufgabe:

- Aktivieren Sie in Ihrer OpenCL-Queue das Flag für *out-of-order-execution*, und verwenden Sie Events, um die notwendigen Abhängigkeiten der Daten-Transfers und Kernel-Executions sicherzustellen.
- Erweitern Sie Ihre Lösung so, dass die Anzahl der Feldelemente, die jedes Workitem im Kernel bearbeitet (bisher 256), flexibel gewählt werden kann.
- Benutzen Sie die Profiling-Möglichkeiten der Events, um herauszufinden, welche Elementzahl pro Workitem für Ihre Karte optimal ist. (Benutzen Sie dazu die Differenz zwischen Startzeit des Kernels **`calcStatistic`** und Endzeit des Kernels **`reduceStatistic`**). Steht das Ergebnis in Verbindung mit der Anzahl der Compute-Units, die Ihre Karte besitzt?