

Dokumentation der Prüfungsaufgaben (Bachelor)

Programming Massively Parallel Processors

Prof. Dr. Jens-Uwe Hahn
WS20/21

Autor: Aydin Mirzaghayev (am180)

Datum: 19.02.2021

Inhaltsverzeichnis

Inhaltsverzeichnis.....	2
1. Aufgabe: Präfixsumme.....	3
Basisalgorithmus.....	3
Umsetzung.....	4
Vielfaches von 256.....	5
Feldlänge(A) \leq 256.....	5
Feldlänge(A) $>$ 256.....	5
Summe aus den letzten Elementen des A- und B- in C-Block ablegen.....	6
Feld D mit allen Werten aus B addieren und in E ablegen.....	6
2. Aufgabe.....	7
a) ImageFX - Convolution.....	7
b) Canny-Algorithmus.....	7

1. Aufgabe: Präfixsumme

Mittels Parallel Processing auf der GPU soll die Präfixsumme eines beliebigen EingabeArrays ermittelt werden.

Präfixsummen Beispiel

Folge	3	2	1	2	1	4	3	2	4	3
PräfixSummen	0	3	5	6	8	9	1	6	18	22

Basisalgorithmus

Der Basisalgorithmus liefert für die festgelegte Länge eines einzigen Feldes die Präfixsumme.

```
int size = 60; // Array Größe
cl_int* input = new cl_int[size]; // Array deklarieren, Größe festlegen

for (int i = 0; i < size; i++) // durch einzelne Elemente iterieren
    input[i] = 1; // dem Element einen Wert zuweisen

praeifixsumme(input, output, size, mgr);
```

Die Funktion **praeifixsumme()** erwartet bei einem Aufruf vier Parameter:

- Referenz zum Array mit Eingabewerten (1, 1, 1, 1, 1, 1, 1, 1, ...)
- Ausgabearray um die Präfixsummen eines jeden Schritts zu hinterlegen
- Länge des Eingabearrays
- OpenCL Manager Objekt

```
int praeifixsumme(cl_int* input, cl_int* output, int size, OpenCLMgr& mgr) {}
```

Sowohl für das Eingabe- als auch für das Ausabefeld, werden Buffers erzeugt **clCreateBuffer()**, um temporär Daten zu hinterlegen. Damit das Algorithmus Daten verarbeiten kann wird in das EingabeBuffer das Eingabearray geschrieben **clEnqueueWriteBuffer()**.

Zum Berechnen der Präfixsumme besteht der Kernel aus zwei Argumenten:

```
__kernel void praeifixsumme256_kernel(__global int* in, __global int* out) {}
```

Die beiden Buffer werden als Argumente jeweils an den Kernel **clSetKernelArg()** übertragen.

```
clSetKernelArg(mgr.praefixsumme256_kernel, 0, sizeof(cl_mem), (void *)&inputBuffer);
clSetKernelArg(mgr.praefixsumme256_kernel, 1, sizeof(cl_mem), (void *)&outputBuffer);
```

Die Ausführung des Kernels findet über **clEnqueueNDRangeKernel()** statt.

```
clEnqueueNDRangeKernel(mgr.commandQueue, mgr.praefixsumme256_kernel, 1, NULL,
global_work_size, local_work_size, 0, NULL, NULL);
```

Nach der Ausführung des Prozesses kann der AusgabeBuffer gelesen werden, welcher das Ergebnis enthält und in der Referenzvariable *output abspeichert.

```
clEnqueueReadBuffer(mgr.commandQueue, outputBuffer, CL_TRUE, 0, size * sizeof(cl_int),
output_b, 0, NULL, NULL);
```

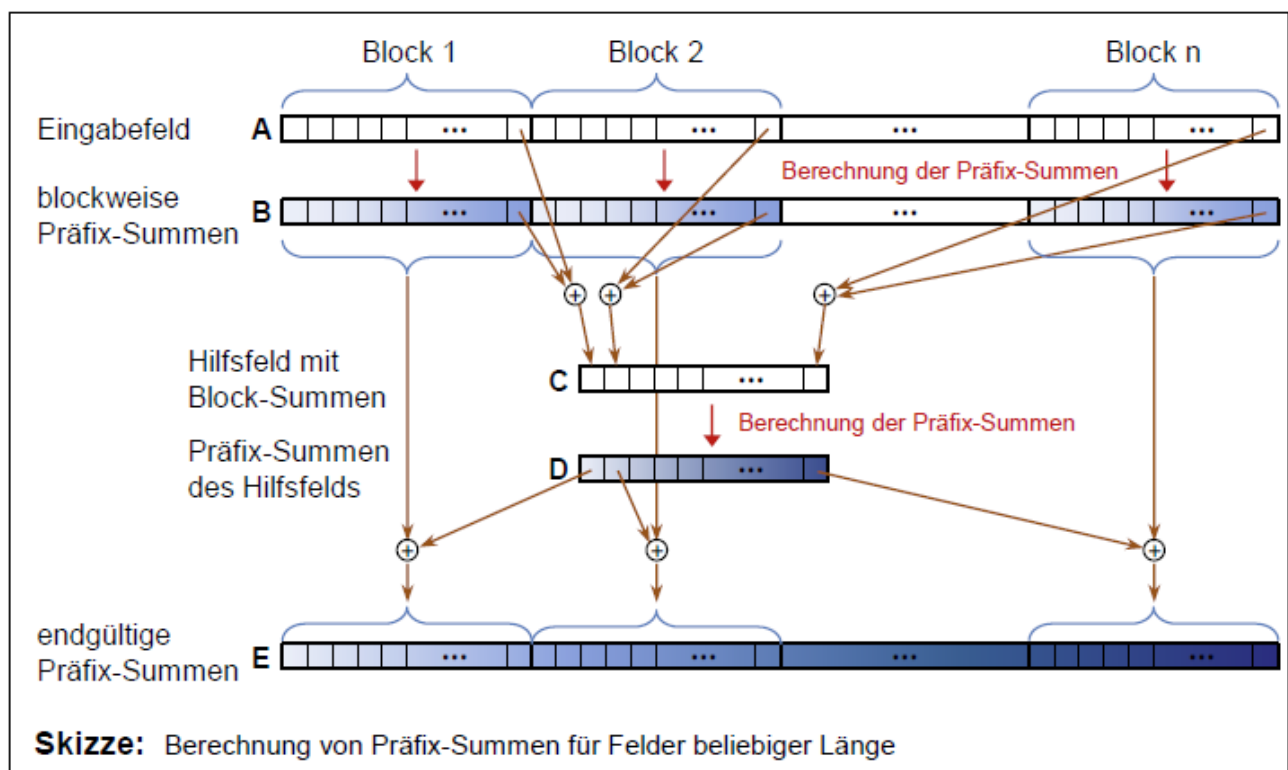
Eingabe	1	1	1	1	1	1	1	1	1	1	1	1
Ausgabe	0	1	2	3	4	5	6	7	8	9	10	11

Nach abgeschlossener Berechnung, sollen der Speicher für die Buffer wieder freigegeben werden.

```
clReleaseMemObject(inputBuffer);
clReleaseMemObject(outputBuffer);
```

Umsetzung

Für die Realisierung der Aufgabe sind weitere dynamische Datentypen zum Hinterlegen von Zwischenergebnissen nötig (siehe Skizze aus der Aufgabenstellung).



```
int size = 60;
cl_int* input = new cl_int[size];           // EingabeArray
cl_int* output_b = new cl_int[size];         // Prefixsummen des Eingabeblocks A
cl_int* output_c = new cl_int[size];         // Summe des letzten Block-Elements aus A und B
cl_int* output_d = new cl_int[size];         // Prefixsummen des Blocks C
cl_int* output_e = new cl_int[size];         // Endergebnis der gesamten Prefixsummen
```

Die Signatur der Funktion **praefixsumme()** muss entsprechend angepasst werden, denn dieser hat nun die Aufgabe die Präfixsummen bis zum Schritt D (siehe Skizze) zu berechnen.

```
int praefixsumme(cl_int* input, cl_int* output_b, cl_int* output_c, cl_int* output_d, int
size, OpenCLMgr& mgr) {}
```

Vielfaches von 256

Um Parallel Processing sinnvoll zu nutzen, ist hier in diesem Schritt notwendig den gesamten Block in kleinere Blöcke zu unterteilen. Jeder Block soll bis zu 256 Elemente aufnehmen können. Falls das Eingabearray 300 Elemente in sich enthält, muss dieser Wert erweitert werden, so dass es ein Vielfaches von 256 ist.

```
int fields = 256;
int size = 300;
int clsize = (size + (fields - 1)) / fields * fields; // = 512
```

Feldlänge(A) <= 256

Dieses Szenario benötigt keine weiteren Anpassung des Basisalgorithmus, denn die Feldlänge ist maximal so groß wie ein Block zulässt.

Feldlänge(A) > 256

Die Buffer werden entsprechend in der notwendigen Größe erzeugt, um darin ein Vielfaches von 256 vergrößerten Felder Platz zu bieten.

```
cl_mem aBuffer = clCreateBuffer(mgr.context, CL_MEM_READ_ONLY, clsize * sizeof(cl_int),
NULL, NULL);
cl_mem bBuffer = clCreateBuffer(mgr.context, CL_MEM_READ_WRITE, clsize * sizeof(cl_int),
NULL, NULL);
cl_mem cBuffer = clCreateBuffer(mgr.context, CL_MEM_READ_WRITE, clsize * sizeof(cl_int),
NULL, NULL);
cl_mem dBuffer = clCreateBuffer(mgr.context, CL_MEM_READ_WRITE, clsize * sizeof(cl_int),
NULL, NULL);
```

Anschließend werden die Buffer als Argument an den Kernel übertragen.

```
clSetKernelArg(mgr.praefixsumme256_kernel, 0, sizeof(cl_mem), (void *)&aBuffer);
clSetKernelArg(mgr.praefixsumme256_kernel, 1, sizeof(cl_mem), (void *)&bBuffer);
clSetKernelArg(mgr.praefixsumme256_kernel, 2, sizeof(cl_mem), (void *)&cBuffer);
clSetKernelArg(mgr.praefixsumme256_kernel, 3, sizeof(cl_mem), (void *)&dBuffer);
```

Die Buffer-Ergebnisse werden nach der Ausführung des Kernels

clEnqueueNDRangeKernel() in den output-Variablen, die wir zu Beginn deklariert haben, gespeichert.

```
clEnqueueReadBuffer(mgr.commandQueue, bBuffer, CL_TRUE, 0, size * sizeof(cl_int), output_b,
0, NULL, NULL);
clEnqueueReadBuffer(mgr.commandQueue, cBuffer, CL_TRUE, 0, size * sizeof(cl_int), output_c,
0, NULL, NULL);
clEnqueueReadBuffer(mgr.commandQueue, dBuffer, CL_TRUE, 0, size * sizeof(cl_int), output_d,
0, NULL, NULL);
```

Summe der letzten Elemente des A- und B-Blocks in C-Block ablegen

Um Parallel Processing sinnvoll zu nutzen, ist hier in diesem Schritt notwendig den gesamten Block in kleinere Blöcke zu unterteilen. Je Block soll bis zu 256 Elemente aufnehmen können. Falls das Eingabearray 300 Elemente in sich enthält, muss dieser Wert erweitert werden, so dass es ein Vielfaches von 256 ist.

Feld D mit allen Werten aus B addieren und in E ablegen

Um Parallel Processing sinnvoll zu nutzen, ist hier in diesem Schritt notwendig den gesamten Block in kleinere Blöcke zu unterteilen. Je Block soll bis zu 256 Elemente aufnehmen können. Falls das Eingabearray 300 Elemente in sich enthält, muss dieser Wert erweitert werden, so dass es ein Vielfaches von 256 ist.

```
int fields = 256;
int size = 300;
int clsize = (size + (fields - 1)) / fields * fields;           // = 512
```

2. Aufgabe

a) ImageFX - Convolution

Für die Realisierung der Aufgabe sind weitere Arrays zum Hinterlegen von Zwischenergebnissen nötig (siehe Skizze aus der Aufgabenstellung).

b) Canny-Algorithmus

Für die Realisierung der Aufgabe sind weitere Arrays zum Hinterlegen von Zwischenergebnissen nötig (siehe Skizze aus der Aufgabenstellung).