Dokumentation der Prüfungsaufgaben (Bachelor)

Programming Massively Parallel Processors

Prof. Dr. Jens-Uwe Hahn
WS20/21

Autor: **Aydin Mirzaghayev (am180)**

Datum: **21.02.2021**

Inhaltsverzeichnis

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

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

Die Funktion **praefixsumme()** 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 praefixsumme(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 praefixsumme256_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);
```

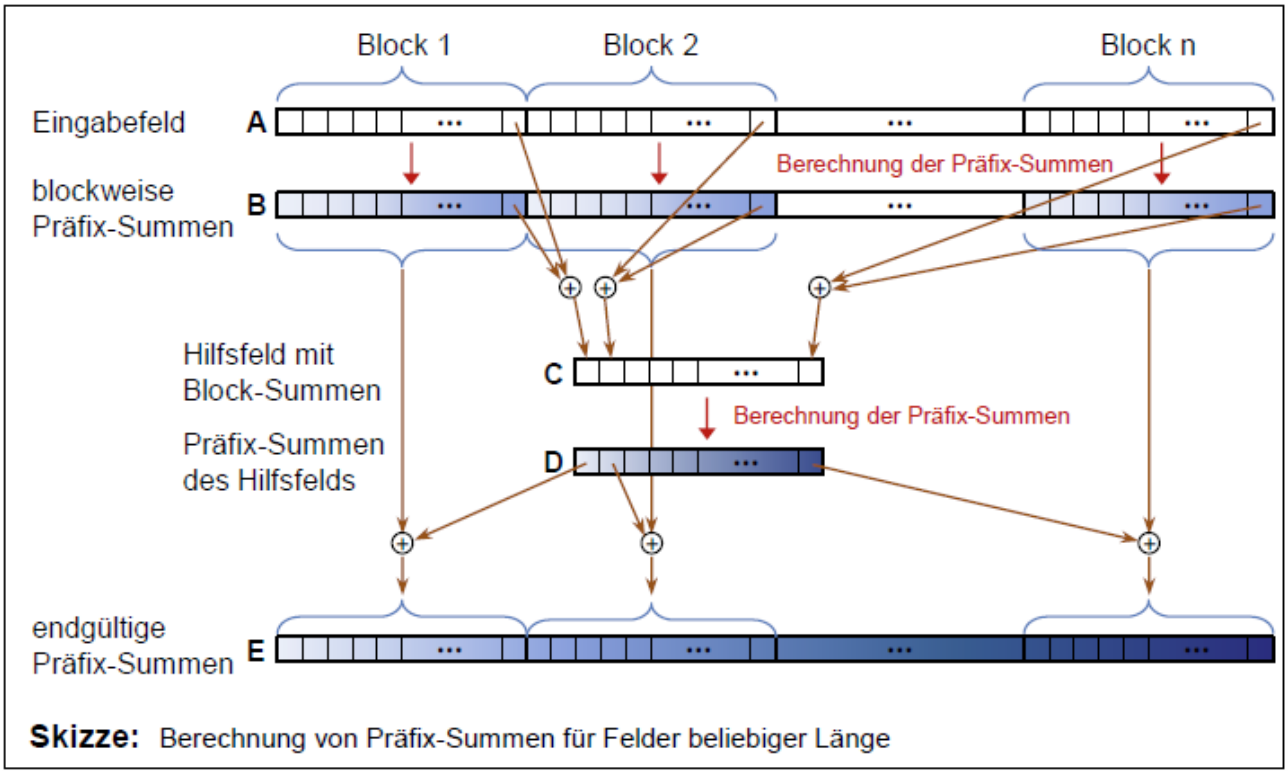
Eingabe	1	1	1	1	1	1	1	1	1	1	1	1
Ausagbe	0	1	2	3	4	5	6	7	8	9	10	11

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).



Skizze: Berechnung von Präfix-Summen für Felder beliebiger Länge

```
int size = 60;
cl_int* input = new cl_int[size];          // EingabeArray
cl_int* output_b = new cl_int[size];       // Pfäfixsummen 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];       // Pfäfixsummen des Blocks C
cl_int* output_e = new cl_int[size];       // Endergebnis der gesamten Präfixsummen
```

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);
```

Block C

Das Prinzip für die Berechnung im Kernel von Block A nach B kann aus dem Basisalgorithmus 1:1 übernommen werden.
C-Feld wird erst immer befüllt, wenn das Algorithmus an dem 255. Feld (0 – 255) eines jeden Blocks ankommt. Die Summe aus dem letzen Feld des A-Blocks und B-Blocks werden addiert und dem C zugewiesen.

```
if (lid == (BLOCK_SIZE-1)) {
	int a_last_item = in[gid];
	int b_last_item = b_out[gid];
	int c_ergebnis = a_last_item+b_last_item;
	c_out[groupid] = c_ergebnis;
}
```

Block D

Block D bildet die Präfixsummen der Felder aus Block C.

```
for (int i = 1; i <= BLOCK_SIZE; i++) {
	int c_last_item = c_out[i-1];
	int d_last_item = d_out[i-1];
	d_out[i] = c_last_item + d_last_item;
	barrier(CLK_LOCAL_MEM_FENCE);
}
```

Blocke E

Im separatem Kernel wird der letzte Block E bestimmt. Dieser resultiert sich aus der Addition eines jeden Feldes im Block D mit dem jeweiligen gesamten Block A.

```
__kernel void summe_kernel(__global int* inB, __global int* inD, __global int* out)
{
	int gid = get_global_id(0);
	int lid = get_local_id(0);
	int groupid = get_group_id(0);

	__local int localArrayB[BLOCK_SIZE];
	__local int localArrayD[BLOCK_SIZE];

	localArrayB[lid] = inB[gid];
	localArrayD[lid] = inD[gid];

	barrier(CLK_LOCAL_MEM_FENCE);

	out[gid] = inB[gid] + inD[groupid];
}
```

2. Aufgabe

Leider konnte ich das zur Verfügung gestelltes Programm ***ImageFX*** lokal nicht funktionsfähig konfigurieren, um es zu starten. Aus diesem Grund behandele ich die Aufgabe in theoretischer Form. Zur Beginn der Woche hatte ich diesbezüglich auch Kontakt mit den Tutoren und im Anschluss mit Ihnen gesucht, um das Problem zu beheben, jedoch bin ich da nicht weiter gekommen. Bitte um Verständnis.

a) ImageFX – Faltung (Convolution)

Def. Bei der Faltung berechnet sich jeder Pixel des Ausgangsbildes als gewichtete Summe der Pixel einer Bildnachbarschaft. Die Gewichte sind die Koeffizienten der Filtermaske (des Filterkerns).[1]

Im folgendem Pseudocode lässt sich die Faltung mit wenigen Zeilen darstellen:

```
schleife jeBildZeile in inputBild {
	schleife jePixel in jeBildZeile {

		setze SUMME zu null;

		schleife jeKernelZeile in Kernel {
			schleife jeElement in jeKernelZeile {
				multipliziere jeElementWert mit entsprechendemPixelWert;
				weise ERGEBNIS zu SUMME;
			}
		}

		setze AusgabeBildPixel zu SUMME;
	}
}
```

Unser Algorithmus besteht aus vier verschachtelten Schleifen. Die äußeren beiden Schleifen iterieren über alle Pixel des Eingabe-Bildes. Die inneren beiden Schleifen iterieren über alle Positionen des Kernels und gewichten die Pixel des Eingabebildes mit dessen Werten[2]. Diese gewichteten Werte werden aufaddiert und in das Pixel des Ausgabebildes geschrieben.

1 http://www.gm.fh-koeln.de/~konen/WPF-BV/BV06a.PDF Seite 3
2 https://www.linux-community.de/ausgaben/linuxuser/2011/07/opencl-workshop-teil-1-grundlagen/2/#article_l1

b) Canny-Algorithmus (Kantendetektion)

Der Canny-Algorithmus ist ein robuster Algorithmus zur Kantendetektion. Er gliedert sich in verschiedene Faltungsoperationen und liefert ein Bild, welches idealerweise nur noch die Kanten des Ausgangsbildes enthält.[3]

Schaubild 1: Ausgangsbild links - Ausgabebild rechts

3 https://de.wikipedia.org/wiki/Canny-Algorithmus