Fakultät Ingenieurswissenschaften

BACHELORARBEIT

Implementierung und Laufzeitoptimierung von Bildverarbeitungsalgorithmen in OpenCL für Embedded Systems

vorgelegt von

Maik Schoesau

Seminargruppe: **ME16wA-B**

Betreuer der Hochschule: **Prof. Dr.-Ing. A. Lampe**
Zweitprüfer: **M. Süß, M. Sc.**

Einreichung: **Mittweida, September 2019**

Abstract

Gegenstand der vorliegenden Arbeit ist die Implementierung und Laufzeitoptimierung von Bildverarbeitungsalgorithmen in OpenCL. Es wird untersucht, wie viel Zeit die Algorithmen zur Bearbeitung von Bildern in Anspruch nehmen und wie stark diese gedrosselt werden kann. Dabei werden verschiedene Speicherbereiche von GPUs und unterschiedliche Rechenmethoden hinsichtlich ihrer Performance beleuchtet. Die Laufzeit der jeweiligen Programmversion wird gemessen, den anderen Varianten gegenübergestellt und ausgewertet. Aus der Untersuchung geht hervor, dass die Laufzeiten der Programme auf bis zu einem Drittel der unbearbeiteten Algorithmen gesenkt werden können.

Inhaltsverzeichnis

1 Aufgabenstellung

Im Rahmen des Praxismoduls im sechsten Semester des Bachelorstudiums waren Bildverarbeitungsprogramme in OpenCL zu implementieren, auf embedded systems zu testen und die Laufzeiten aufzuzeichnen. Es wurden Programme wie der Sobel- und Gauß-Filter sowie der SURF-Algorithmus in verschiedenen Varianten implementiert und die Laufzeit zunächst auf einem Laptop gemessen.

Anschließend war die Aufgabe der vorliegenden Arbeit die Laufzeit des SURF-Algorithmus zu optimieren und die Laufzeit des Gauß-Filter-Programms für verschiedene Filtergrößen zu messen und auszuwerten.
Danach sollten die verfassten Programme auf ein OpenCL-fähiges embedded system aufgespielt, getestet und deren Laufzeiten analysiert werden.
Abschließend waren die Laufzeiten vom embedded system denen des Laptops gegenüberzustellen und ebenfalls auszuwerten.

2 Einführung OpenCL

Abbildung 1: OpenCL Logo[1]

OpenCL (Open Computing Language) ist ein offener, lizenzfreier Standard für verschiedene Plattformen und Parallele-Programmierung für diverse Prozessoren in PCs, Servern, mobilen Geräten und embedded systems. OpenCL erhöht die Geschwindigkeit und Reaktionsbereitschaft für ein breites Spektrum von Anwendungen wie Gaming, neuronale Netze und wissenschaftliche bzw. medizinische Software. [1]
OpenCL zeichnet sich durch die Lauffähigkeit auf verschiedenen Plattformen aus, was

[1] Quelle: https://upload.wikimedia.org/wikipedia/commons/b/b8/OpenCL.jpg

den Vorteil mit sich bringt, den Quellcode nur einmal verfassen zu müssen, um das Programm auf verschiedenen Systemen anwenden zu können. Des Weiteren kann OpenCL mehrere Geräte (CPUs und GPUs) zeitgleich ansprechen und hat eine standardisierte Vektorverarbeitungsroutine, welche ANSI C/C++ nicht bereitstellt. Zusätzlich deckt der Standard Aspekte der simultanen Abarbeitung ab, in dem ein einzelnes Bearbeitungselement (*processing element*) seine Ressourcen unter Prozessen und Threads teilt. Der wichtigste Aspekt für OpenCL ist jedoch die parallele Programmierung, die es ermöglicht mehrere Operationen an verschiedene ansprechbare Geräte (*devices*) zu delegieren und zeitgleich ausführen zu lassen. Diese Rechenaufgaben werden *Kernels* genannt. [2] Jene Delegation und zeitgleiche Ausführung von Kernels führt zu einer stark erhöhten Rechengeschwindigkeit und ist für die Reduzierung der Ablaufzeiten in verschiedensten Systemen essenziell.

3 Funktionsweise einer GPU

Um die Verbindung vom Computer zur Außenwelt herzustellen, wird im Normalfall auf eine grafische Ausgabe am Monitor zurückgegriffen. Die Grafikkarte des Computers berechnet und steuert diese visuelle Darstellung. Dabei leitet der Prozessor bei der Ausführung eines Programms die berechneten Daten an jene Grafikkarte weiter und wandelt sie dort so um, dass der Monitor oder ein anderes Ausgabegerät, wie z.B. Beamer, diese sinnig wiedergeben kann. Grafikkarten besitzen heutzutage eigenständige Recheneinheiten mit eigener GPU (Graphics Processing Unit), die speziell bezüglich der Hardwareseite auf Grafikberechnungen spezialisiert sind und somit wesentlich schneller als CPUs in diesem Gebiet arbeiten.
Beim Berechnen einer Grafik arbeitet die GPU eine starre Prozessabfolge (*processing pipeline* oder *graphics pipeline*) ab, welche nachfolgend erläutert wird.

Zum Beginn der Abfolge schickt die CPU aus der aktuell laufenden Anwendung der GPU Daten, die einfache Geometrien beschreiben und zeichnen. Diese werden *Primitive* genannt und können Punkte, Linien und Polygone darstellen.

Anschließend werden die Eck- bzw. Scheitelpunkte (*Vertices*) der Primitive im Schritt *vertex processing* umgeformt. Dabei werden für alle Vertices Normalen über eine invers transponierte Matrix, Vertexpositionen mithilfe von Projektionsmatrizen, Texturkoordinaten mit Texturmatrizen und die Punktgröße errechnet, Lichtkalkulationen durchgeführt und der Farbstatus eingestellt.

Nachdem alle Attribute der Vertices bestimmt sind, sieht die Pipeline den Zusammenbau der Primitive im Schritt *primitive assembly* vor. Hier werden Vertex Daten zu einzelnen Primitiven gesammelt. Punkte brauchen einen Vertex, Linien zwei, Dreiecke drei usw. Dies ist notwendig, da in der nächsten Phase des Ablaufs mit Sätzen von Vertices gerechnet wird und die Operationen von der Art des Primitivs abhängen.

Im nächsten Schritt schneidet zuerst die GPU die Primitive, die letztlich auf dem Bildschirm nicht zu sehen sind, ab. Wenn ein Primitiv sich komplett außerhalb des Sichtbereiches befindet, wird es aussortiert und keine weiteren Berechnungen sind nötig. Wenn ein Primitiv sich partiell innerhalb des Sichtbereiches aufhält, teilt (clippt) die GPU es. Die zweite Operation dieses Schritts berechnet die Fensterkoordinaten für eine gegebenenfalls perspektivische Sicht und überprüft, ob das Primitiv zum Blickpunkt oder von ihm weg zeigt. Die Primitive, die zum Blickpunkt zeigen, werden dann behalten und die von ihm weg zeigen, verworfen.

Darauf folgt die Zerlegung der Primitive in kleinere Einheiten (*Fragmente*), die mit einzelnen Pixeln des Framebuffers vom Ausgabegerät übereinstimmen. Dieser Vorgang wird Rasterung (*rasterization*) genannt. Ein Fragment hält dabei die entsprechenden Fensterkoordinaten und assoziierte Daten wie Farbe und Texturkoordinaten. Die Werte der Attribute jedes Pixels errechnet die GPU durch Interpolation.

Anschließend werden die Fragmente im Schritt *fragment processing* bearbeitet. In dieser Phase werden die Texturkoordinaten, die mit dem jeweiligen Fragment assoziiert sind, genutzt, um auf den Texturspeicherbereich des Grafikspeichers zugreifen zu können. Diese Operation wird *texture mapping* genannt. Andere Berechnungen dieses Schritts sind unter anderem FOG und COLOR SUM. Bei FOG berechnet die GPU die Farbe des Fragments in Abhängigkeit vom Abstand zum Blickpunkt und bei COLOR SUM die Kombination der Werte von der primären und sekundären Farbe des Fragments.

Danach folgt die Sequenz *per-fragment-operations*. In dieser wird beispielsweise abgefragt, ob ein Pixel sichtbar ist oder von einem anderen Fenster verdeckt ist. Vermischungs-, Dithering- und logische Operationen werden auch von diesem Schritt abgedeckt. Diese Berechnungen haben einen geringen Rechenaufwand und sind einfach auf der Hardware zu implementieren.

Zum Schluss folgen Einstellungen, die den *Framebuffer* betreffen. Dieser Schritt wird *framebuffer operations* genannt und beinhaltet double buffering, stereo images und ähnliches.
Nachdem der Framebuffer beschrieben ist, wird dieser ausgelesen und auf dem Ausgabegerät angezeigt.

Eine kleine Übersicht der Pipeline ist in der folgenden Abbildung dargestellt.

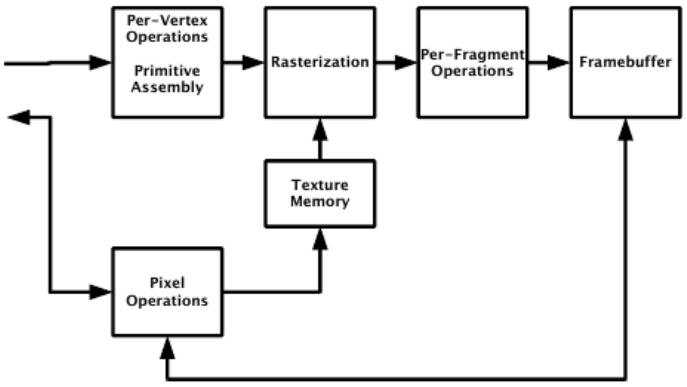

Abbildung 2: graphics pipeline[2]

4 Programmiermodell OpenCL

Ein OpenCL-Programm besteht grundsätzlich aus zwei Teilen: einem *Hostprogramm* und einem oder mehreren Kernelprogrammen. In dem Hostprogramm, welches in jeder Programmiersprache mit einer OpenCL-API (Application Programming Interface) geschrieben werden kann, wird eine Reihe von Datenstrukturen verarbeitet, die nachfolgend erläutert werden. Die Hostprogramme dieser Abschlussarbeit wurden in der Programmiersprache C verfasst.

Zunächst muss die *Plattform* (`cl_platform_id`) des Systems bestimmt werden. Diese Datenstruktur abstrahiert die Treiber für die OpenCL-Implementierung und ermittelt, welche OpenCL-Schnittstelle das System besitzt. Das Ermitteln dieser Schnittstelle, die jeder Hersteller unterschiedlich konfiguriert, ist notwendig, um das verfasste Programm für verschiedene Systeme von unterschiedlichen Herstellern kompatibel zu gestalten. Würde dies nicht implementiert, könnte die universelle Lauffähigkeit des Programms nicht sichergestellt werden, was einen der großen Vorteile von OpenCL liquidieren würde.

Im Anschluss sucht man nach installierten Geräten (devices), die OpenCL- kompatibel sind und die gleiche OpenCL-Schnittstelle nutzen. Die zu beschreibende Datenstruktur lautet `cl_device_id`. An diese Geräte werden später die Kernels mit den Berechnungsaufgaben geschickt und verarbeitet.

Nachdem die verfügbaren Geräte erfasst sind, wird ein *Kontext* (`cl_context`) erstellt. Dieser sammelt alle gewünschten Geräte, die die Kernel verarbeiten sollen und ermöglicht eine Anweisungsreihe (*command queue*), die eine Versendung der Rechenaufgaben vom Host an die ausgewählten Geräte erlaubt. Es können mehrere Kontexte erstellt werden

[2] Quelle: https://www.khronos.org/registry/OpenGL/specs/es/2.0/es_full_spec_2.0.pdf, S.13

und ein einzelnes Gerät kann dabei in verschiedenen Kontexten verwendet werden.

Danach wird die Datenstruktur *Programm* (`cl_program`) erstellt, welche als eine Art Behältnis für Kernel verstanden werden kann. Diese Datenstruktur beinhaltet, wie auch die Kernel, ausführbare Kommandos. Jedoch werden im Programm die gesamten Kernels und nicht wie in den Kernelfunktionen nur die einzelnen Rechenaufgaben, die auf den Geräten ausgeführt werden, aufbewahrt.

Nachdem die Datenstruktur `cl_program` erstellt ist, muss das Programm kompiliert werden. Die Funktion `clBuildProgram` übersetzt und verbindet `cl_program` mit den Geräten, die mit der Plattform assoziiert sind.

Anschließend werden die Programmfunktionen mithilfe des Befehls `clCreateKernel` in Kernel-Datenstrukturen (`cl_kernel`) gepackt, wovon der Host sie später in eine Anweisungsreihe aufnimmt und an das jeweilige Gerät verschickt.

Bevor die Kernel jedoch an die Geräte geschickt und ausgeführt werden können, müssen die Daten, die zu bearbeiten sind (Bilder, Arrays, Matrizen, etc.), der Funktion zur Verfügung gestellt werden. Diese werden *Argumente* genannt und brauchen einen reservierten Speicherbereich für die zu bearbeitenden Daten, welcher vor dem Setzen der Kernelargumente mit dem Befehl `clCreateBuffer` oder `clCreateImage` aktiviert wird. Wenn Pixelwerte dem Kernel zu übergeben sind, wird ein *Image-Objekt* verwendet, was durch `clCreateImage` erstellt wird. In jedem anderen Fall wird auf ein Puffer-Objekt (*buffer-object*) zurückgegriffen. Um den Zugriff des Kernels auf die Daten beider Speicher-Objekte zu gewährleisten, muss der Kontext, die Art und Größe des Speichers, ein Zeiger auf die Daten und ein *flag* gesetzt werden, welches die Art der Lesbarkeit der Daten für den Kernel bestimmt (`cl_mem_flags options`). Dabei werden die Lese- und Schreibrechte des Kernels, sowie die Art der Speicherplatzreservierung des Speicher-Objekts im Hostprogramm eingestellt. Es gibt hierfür folgende Einstellmöglichkeiten: Lesen von Daten vom Gerät zum Host, Schreiben von Daten vom Host zum Gerät und Kopieren von Daten zwischen Geräten. Beim Image-Objekt kommen noch die Argumente `desc` und `cl_image_format` dazu. Das Image-Format bestimmt, welche Kanalanordnung und welchen Datentyp das Bild besitzt. Desk stellt unter anderem ein, über wie viele Dimensionen das Image verfügt und wie groß die jeweiligen Abmaße sind. Das folgende Programmfragment erzeugt beispielsweise ein Schwarz-Weiß-Bild, welches 2 Dimensionen besitzt und dessen Pixelhelligkeitswerte im Typ `CL_UNORM_INT8` angegeben werden. Die Breite und Höhe des Bildes sind von den Variablen `width` und `height` abhängig.

```
png_format_gray.image_channel_order = CL_LUMINANCE;
png_format_gray.image_channel_data_type = CL_UNORM_INT8;

desc.image_type = CL_MEM_OBJECT_IMAGE2D;
desc.image_width = width;
desc.image_height = height;
```

Der Datentyp bildet nicht vorzeichenbehaftete, normalisierte 8-Bit Ganzzahl (integer) ab. Das bedeutet, dass der Datentyp Zahlen von 0.0 bis einschließlich 1.0 darstellen kann. Die Farbwerte eines Bildes werden über Datentypen mit vier Kanälen bestimmt, die wie Arrays aufgebaut sind (z.B. `float4`). Dabei entscheidet die Option `image_channel_order` der Image-Format-Variable darüber, wie diese Kanäle belegt sind. Im Programmfragment ist mit `CL_LUMINANCE` eine Kanalbelegung ausgewählt worden, welche das Bild in Grauwerten darstellt. In dieser Einstellung bestimmt nur der erste Kanal über die Farbe des jeweiligen Bildpunktes und die restlichen Drei werden auf 1.0 gesetzt, wie das folgende Beispiel zeigt.

```
float4 Pixelfarbwert = (float4)
                       (Farbwert_Kanal1, 1.0f, 1.0f, 1.0f)
```

Weitere Kanalbelegungsarten sind u.a. `CL_RGBA` und `CL_RGB`. Diese können die Bildpunkte auch in Farbe darstellen.

Nachfolgend werden die Argumente mit dem Befehl `clSetKernelArg`, die Kernel an eine Anweisungsreihe (*command queue*) angehangen und anschließend zu einem Gerät geschickt. Jene Anweisungsreihe ermöglicht erst die Kommunikation zwischen Host und Gerät, da diese den Kernel mit dem Kontext und dabei mit dem Gerät assoziiert.

Danach wird ein Kernel-Ausführungskommando an das gewünschte Gerät gesendet und der Kernel ausgeführt. Dafür gibt es verschiedene Ausweisungen. In den Programmen in dieser Arbeit wurde stets der Kernel mit dem Befehl `clEnqueueNDRangeKernel` gestartet, da dieser die Möglichkeit bietet, die *Dimensionen* des Kernels zu setzen. Ähnlich wie bei ineinander geschriebenen for-Schleifen, kann über verschiedene Argumente der Funktion die Anzahl der Laufvariablen und deren Iterationen festgelegt werden. Diese stimmen in der Regel mit der Menge der Eingangs- oder Ausgangsdaten überein. Die Anzahl der Dimensionen wird über das Argument `work_dims` der Funktion festgelegt und ist der Anzahl der for-Schleifen gleichzusetzen. Die Iterationen in der jeweiligen Dimension werden Arbeitselemente (*work-items*) genannt und korrespondieren mit den Laufvariablen der ineinander verschachtelten for-Schleifen. Diese werden in der Funktion `clEnqueueNDRangeKernel` über das Argument `global_work_size` gesetzt und im Kernel über `get_global_id(Nummer_der_Dimension)` aufgerufen.

Nachdem alle work-items und somit der komplette Kernel auf dem Gerät abgearbeitet wurde, können die verarbeiteten Daten aus dem Pufferspeicher über den im Hostprogramm verwendeten den Befehl `clEnqueueReadBuffer` oder `clEnqueueReadImage` auf die CPU gelesen und weiterverarbeitet werden.

Abgeschlossen wird ein OpenCL-Programm mit dem Löschen der beschriebenen Register wie Pufferspeicher, Pixelarrays und Datenstrukturen wie Plattform, Kernel und Kontext.

4.1 Plattform-Modell

Wie bereits im Abschnitt 3 erwähnt, ist ein Teil eines OpenCL-Programms das Hostprogramm, in dem eine CPU (Host) u. a. den Hersteller (Plattform) bestimmt und die Ausführung der verschiedenen Rechenschritte auf den OpenCL-kompatiblen Geräten (devices) koordiniert. Das OpenCL Plattform-Model definiert dabei im Standard ein Gerät als ein Array von Recheneinheiten (*compute units*), welche funktionell unabhängig voneinander agieren. Die Recheneinheiten werden weiter in Verarbeitungselemente (processing elements) unterteilt. Abbildung 3 zeigt grafisch diese Hierarchie.

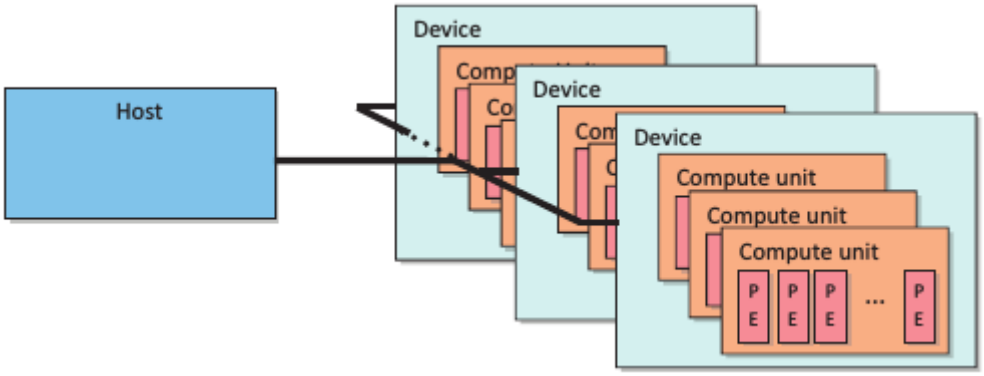

Abbildung 3: Plattform-Model[3]

Die Geräte besitzen eine Hardwarearchitektur, die mit der Architektur der Programmiersprache OpenCL übereinstimmt. So besitzt beispielsweise eine AMD Radeon 7970 Grafikkarte (device) 32 Vektorprozessoren (compute units), welche in je 64 Verarbeitungselemente (processing elements) unterteilt sind.

4.2 Speicher-Modell

Die Speicherarchitekturen unterschiedlicher Grafikkarten variieren hinsichtlich ihres Aufbaus und der Größe der einzelnen Komponenten signifikant. Durch ein abstraktes, im Standard definiertes Speichermodell ist die Übertragbarkeit der OpenCL-Programme auf verschiedene Fabrikate bei gleichzeitiger Ausschöpfung der potenziellen Arbeitsleistung gewährleistet. Dieses Modell wird dann im Programm angesprochen und der Treiber der jeweiligen Plattform mapped die Speicherbereiche auf die entsprechende Hardware. Bei der Deklaration von Variablen und der Übergabe der Argumente im Kernelprogramm werden die im Standard definierten Schlüsselwörter verwendet und assoziieren damit die Variable mit dem entsprechenden Speicherbereich und dessen speziellen Eigenschaften.

Es gibt vier verschiedene Adressbereiche: *global memory*, *constant memory*, *local memory* und *private memory*. Der globale Adressbereich (global memory) speichert Daten für das komplette Gerät. Er wird mit `__global` initialisiert und kann von allen work-items

[3]Quelle: Benedict R. Gaster: Heterogeneous Computing with OpenCL, Morgan Kaufmann, 2012, S.20

gelesen und beschrieben werden. Er hat die langsamste Zugriffsdauer für work-items und stellt in der Regel den größten Speicherbereich zur Verfügung, welcher vor allem bei Pufferspeichern (Puffer- und Image-Objekten) Verwendung findet.

Der Konstantenspeicher (constant memory) ist für Daten ausgelegt, die alle work-items gleichzeitig brauchen (z.B. Naturkonstanten wie π oder e). Dieser Adressbereich wird mit `__constant` aufgerufen und ist, wie in Abbildung 4 dargestellt, ein Teil des globalen Speicherbereiches.

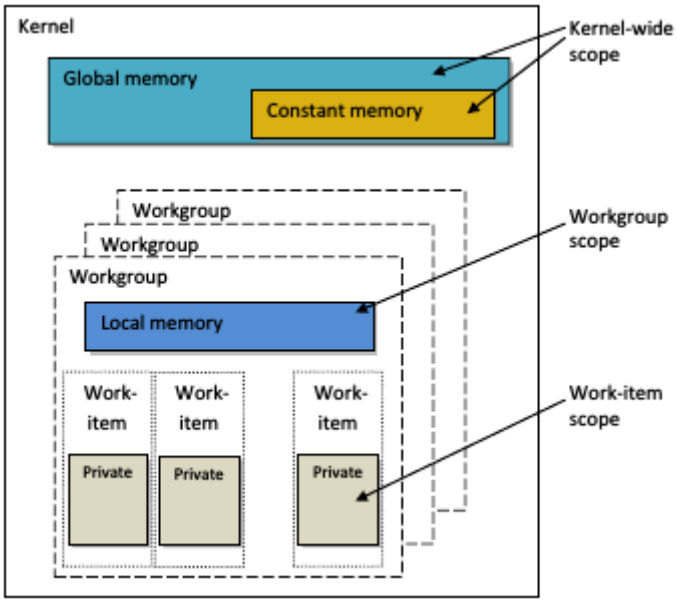

Abbildung 4: Speicherbereiche[4]

Der lokale Adressbereich (local memory) speichert Daten für work-items in einer Arbeitsgruppe (*work-group*) und wird mit `__local` aufgerufen. Dieser Speicherbereich ist wesentlich kleiner als der globale, aber die Zugriffsgeschwindigkeit ist wesentlich höher ($\approx$100 mal schneller) und findet aufgrund dessen vor allem für Zwischenwerte der work-items Verwendung. [3]
Work-groups sind eine Art Behältnis für work-items, die eine Recheneinheit (compute unit) zeitgleich ausführt. Work-items derselben work-group können zudem über *Barrieren* (`fence` und `barrier`) synchronisiert werden. Die Größe einer work-group wird über das Argument `local_work_size` der Funktion `clEnqueueNDRangeKernel` festgelegt und erfordert die Berücksichtigung der zugrundeliegenden physischen Hardware der GPU und der dimensionalen Größe der Berechnungsaufgabe. Um die beste Laufzeit des Programms zu erreichen, ist es wichtig, so viele work-items wie möglich in einer work-group unterzubringen, da nur diese den schnellen lokalen Adressbereich nutzen und den langsameren globalen Adressbereich umgehen können. Die maximal mögliche Größe der

[4]Quelle: Benedict R. Gaster: Heterogeneous Computing with OpenCL, Morgan Kaufmann, 2012, S.29

work-groups ist durch die Hardware festgelegt und bleibt unabhängig von der Anzahl der work-items des Kernels erhalten. Falls die Anzahl der work-items (`global_work_size`) die maximale Größe der work-groups übersteigt, wird eine weitere Arbeitsgruppe erstellt. Die maximale Anzahl der work-groups in einem Programm ist hingegen nahezu unbegrenzt. Wie viele work-groups jedoch gleichzeitig verarbeitet werden können, hängt von der Anzahl der Recheneinheiten der GPU ab. Die Größe der work-groups der jeweiligen Anwendung hängt von den zur Verfügung gestellten Ressourcen des Geräts und der Ressourcen, die der Kernel auf den Gerät einnimmt, ab. Der lokale und private Speicher bestimmt dabei hauptsächlich die Ressourcen, die der Kernel belegt und damit die Anzahl der maximal verfügbaren work-items in der work-group. Je mehr Speicher der Kernel belegt, desto weniger work-items können pro work-group abgearbeitet werden. Um die optimale work-group-Größe für den vorliegenden Kernel und das betreffende Gerät zu bestimmen, stellt der OpenCL-Standard die Option `CL_KERNEL_WORK_GROUP_SIZE` der Funktion `clGetKernelWorkGroupInfo` bereit.

Das folgende Beispiel soll die Auswahl der work-group-Größe veranschaulichten. Eine Temperaturmessreihe, bestehend aus 150 Werten, soll geglättet werden. Zum berechnen des Programms wird die in Abschnitt 4.1 aufgeführte Grafikkarte verwendet. Diese besitzt 32 compute units mit jeweils 64 processing elements, was bedeutet, dass 64 Werte pro work-group verarbeitet werden können. Deshalb werden drei work-groups erstellt, wovon zwei die volle work-group-Größe von 64 Werten ausschöpfen können und die Dritte die restlichen 22 Werte aufnimmt. Das Array, in dem die Messwerte gespeichert sind, wird in drei Teile (64, 64 und 22) zerlegt und auf drei compute units aufgeteilt. Um die Messreihe zu glätten, liest das Programm in jeder Recheneinheit den vorgehenden, nachfolgenden und aktuellen Wert der betrachteten Position aus, summiert diese und dividiert das Ergebnis durch 3. Der errechnete Quotient wird an die entsprechende Position im Array geschrieben, stellt den Durchschnitt der drei Messwerte dar und führt zu einer Dämpfung der Messspitzen. Nachdem alle Positionen im Array geglättet sind, werden die Ergebnisse dem Host zurückgeschickt und die geglättete Messreihe weiterverarbeitet.

Der private Adressbereich (private memory) ist nur für jedes work-item exklusiv verfügbar und wird mit `__private` initialisiert. Dieser Speicherbereich ist kleiner als alle anderen und dafür am schnellsten zu erreichen. Kernelargumente ohne Präfix und alle Variablen, sowie Nichtkernel-Funktionen sind per Definition dem privaten Speicherbereich zugeordnet und stellen die Hauptverwendung dar.

Die beschriebenen Speicherbereiche von OpenCL ähneln stark der Architektur aktueller Grafikkarten. In Abbildung 5 ist die Verknüpfung zwischen den Adressbereichen von OpenCL und einer AMD Radeon 7970 Grafikkarte dargestellt.

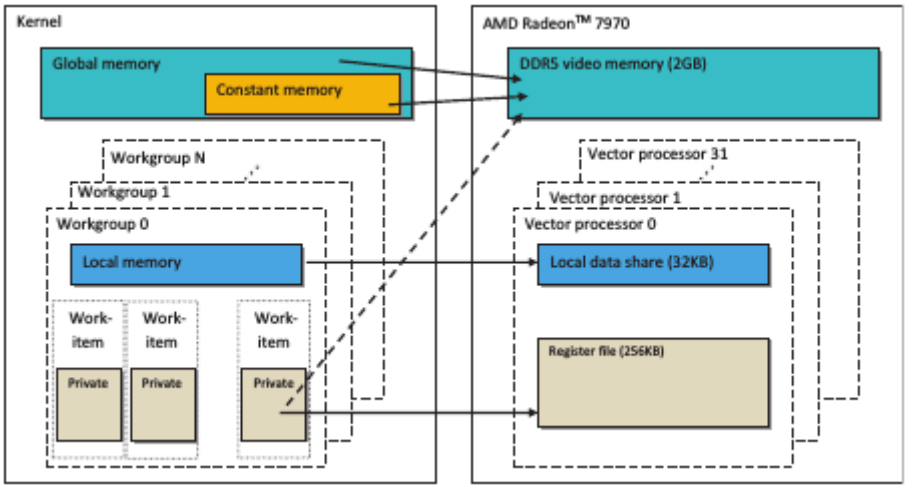

Abbildung 5: Speichermapping[5]

Kenntnis über die Eigenschaften der einzelnen Speicherbereiche und deren Größe auf der jeweils vorhandenen Grafikkarte sind für eine schnelle Abarbeitung eines OpenCL-Programms auf einer GPU essentiell. Hinzu kommt, dass sich GPUs und CPUs hinsichtlich der Arbeitsleistung einzelner Programmoperationen signifikant unterscheiden. So sind beispielsweise Vergleichs- und Entscheidungsoperationen und die Modulo-Division auf CPUs schneller als auf GPUs gelöst. Erst mit Berücksichtigung der eben genannten Eigenschaften von GPU, CPU und OpenCL, kann die bestmögliche Laufzeit des Programms erreicht werden.

[5]Quelle: Benedict R. Gaster: Heterogeneous Computing with OpenCL, Morgan Kaufmann, 2012, S.30

5 Programme

In den folgenden Kapiteln sind die implementierten Filterprogramme erklärt. Die Ergebnisse der einzelnen Bildbearbeitungsprogramme werden visuell durch bearbeitete Versionen des in Abbildung 6 dargestellten Images repräsentiert. Das folgende Bild des Autos dient als Referenzobjekt.

Abbildung 6: Auto unbearbeitet[6]

5.1 Sobel-Filter

Der Sobel-Filter ist ein Kantendetektierungsfilter, der mithilfe der *mathematischen Faltung* die erste Ableitung der Pixelhelligkeitswerte berechnet und gleichzeitig vertikal zur Ableitungsrichtung glättet. Es wird dabei sowohl horizontal (xx-Richtung) als auch vertikal (yy-Richtung) abgeleitet. Mit dem 3*3-Filter werden hohe Frequenzen im Bild mithilfe von Grauwerten dargestellt. Die größten Intensitäten treten dort auf, wo sich die Helligkeit des Originalbildes am stärksten ändert und somit die schärfsten Kanten befinden. [4]

Im ersten Teil des Sobel-Operators wird jedes Pixel des Originalbildes mit zwei Matrizen gefalten. Die erste Faltung detektiert die horizontalen und die Zweite die vertikalen Kanten, wobei die Reihenfolge der Berechnung keine Rolle spielt. Das Ergebnis wird in der Variable G_x bzw. G_y gespeichert. Gleichung (1) zeigt die Bildungsvorschrift für G_x und Gleichung (2) die für G_y.

$$G_x = S_x * A = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * A \tag{1}$$

[6] Quelle: Source Code for GNU, https://www.manning.com/books/opencl-in-action

$$G_y = S_y * A = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A \tag{2}$$

Im anschließenden und letzten Teil der Berechnung werden beide Zwischenwerte zum finalen Farbwert des Sobel-Filters für das Pixel verrechnet und in der Variable G gespeichert.

$$G = \sqrt{G_x^2 + G_y^2} \tag{3}$$

Bei der Implementierung in OpenCL wird im Hostprogramm ein Kernel mit dem Originalbild als Eingangsargument und einem noch unbeschriebenen, jedoch der Größe des Eingangsbildes entsprechenden, Ausgangsargument angelegt. Sowohl das Eingangs- als auch das Ausgangsargument ist dabei als Image-Objekt deklariert und die Kanalanordnung der Farbwerte auf `CL_LUMINANCE` festgelegt, da es sich um ein Grauwertbild handelt. Weil das Hostprogramm zur Interpretation der Ausgangspixel ein 2D-Bild mit einem 8-Bit Integer Datentyp erwartet, wird darüber hinaus der Kanal-Datentyp auf `CL_UNORM_INT8` gesetzt und in `desc` der Bildtyp `CL_MEM_OBJECT_IMAGE2D` verwendet. Anschließend startet der Kernel, das Hostprogramm liest nach der Berechnung aller Pixel die bearbeiteten Bildpunkte aus dem Image-Objekt und schreibt sie in eine PNG-Datei.

Bei der Erstellung des Kernelprogramms wurde der Quellcode aus dem Buch “OpenCL programming guide“ übernommen und modifiziert. [5]
So wurden die Farbwerte aus x- und y-Richtung vom Datentyp `float4` in `float` geändert, weil ein Schwarz-Weiß-Bild seine Farbwerte nur auf einem Kanal (dem Ersten von vier) speichert. Für die beiden zu Beginn des Kapitels beschriebenen Faltungen ((1) und (2)) werden die acht umliegenden Pixel des betrachteten Bildpunkts ausgelesen und in den Matrizen verrechnet. Die Berechnung für G_x ist im folgenden Programmfragment aufgeführt.

```
float4 p00 = read_imagef(src, sampler, (int2)(x - 1, y - 1));
float4 p10 = read_imagef(src, sampler, (int2)(x, y - 1));
float4 p20 = read_imagef(src, sampler, (int2)(x + 1, y - 1));

float4 p01 = read_imagef(src, sampler, (int2)(x - 1, y));
float4 p21 = read_imagef(src, sampler, (int2)(x + 1, y));

float4 p02 = read_imagef(src, sampler, (int2)(x - 1, y + 1));
```

```
float4 p12 = read_imagef(src, sampler, (int2)(x, y + 1));
float4 p22 = read_imagef(src, sampler, (int2)(x + 1, y + 1));

float gx = -p00.x + p20.x + 2.0f * (p21.x - p01.x) -
           p02.x + p22.x;
```

Am Ende des Kernelprogramms ermittelt die folgende Anweisung das Endergebnis G.

```
float g = native_sqrt(output_gx.x * output_gx.x + output_gy.x
          * output_gy.x);
```

In Abbildung 7 ist das Ergebnis der Berechnungen zu sehen.

Abbildung 7: Auto gefiltert mit Sobel-Filter

Der Quellcode des Programms ist dem Dokument angehangen (Anlage 1).

5.2 Gauß-Filter

Der Gauß-Filter ist ein Filter, der Bilder weichzeichnet und kleinere Strukturen (Verpixelung, Bildfehler, Rauschen) herausfiltert. Spektral entspricht er einem Tiefpassfilter. [6]

Die Berechnung der Bildpunkte erfolgt ähnlich wie beim Sobel-Filter. Jedes Pixel und seine unmittelbaren Nachbarn werden mit einer Matrix gefalten, deren Werte aus der diskreten Impulsantwort der Gaußschen Glockenkurve abgeleitet sind.
Die Impulsantwort lässt sich in zwei Dimensionen mit folgender Formel beschreiben.

$$h(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{4}$$

Die im Programm verwendete 3*3-Matrix ist nachfolgend aufgeführt:

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Es sollten zunächst zwei Varianten implementiert werden. Die erste Variante sollte die Bearbeitung der Pixel mit einem 2D-Filter realisieren und die zweite Variante mit zwei 1D-Filtern. Die Aufteilung auf zwei 1D-Filter ist durch die Separierbarkeit des Gauß-Filters möglich. Beide Programme sollten dabei die Berechnung mit nur einem Kernel ausführen. Für die Implementierung der ersten Variante konnte in großen Teilen das Sobel-Filter-Programm verwendet werden. Es wurden nur die Filterwerte modifiziert und das Auslesen der Pixelwerte, sowie die Faltung in zwei ineinander verschachtelten for-Schleifen umgeschrieben. Das Programm rechnet mit den Werten der eben beschriebenen Matrix, die zusätzlich normalisiert sind, indem jeder Wert der Matrix durch die Summe des Betrags aller Matrixwerte dividiert wurde. Der Rest des Programms stimmt mit dem des Sobel-Filters überein. Das Ergebnis der Berechnung ist in Abbildung 8 dargestellt.

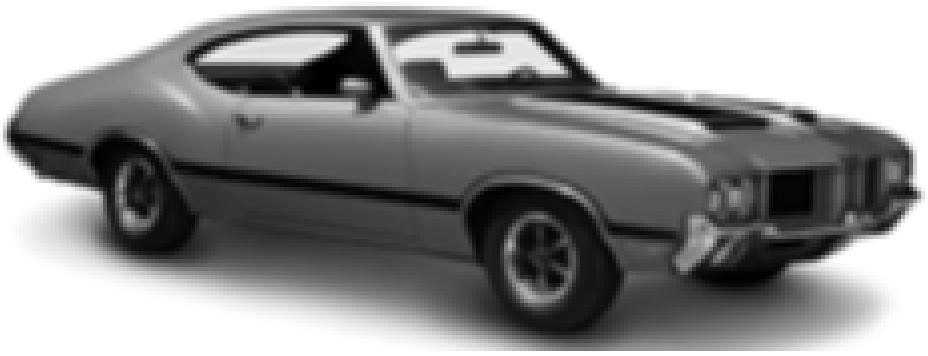

Abbildung 8: Auto gefiltert mit Gauß-Filter

Bei der Implementierung der zweiten Variante konnte auf das Hostprogramm des 2D-Filters und dessen Arbeitsprinzip bei der Faltung zurückgegriffen werden. Der Gauß-Filter wurde in zwei 1D-Filter (horizontaler und vertikaler Filter) separiert und anschließend als Array im Kernel implementiert. Die Umrechnung des Filters ist nachfolgend aufgeführt.

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \tag{5}$$

Die Pixelhelligkeitswerte werden im Programm nach der ersten Faltung in ein Array geschrieben und dort zwischengespeichert. Dabei faltet das Programm mit nur einer for-Schleife, da bei einem 1D-Filter nur eine Laufvariable nötig ist. Mittels einer Barriere sollte das Programm warten, bis alle work-items die erste Berechnung abgeschlossen haben. Die Funktion wird wie folgt aufgerufen:

```
barrier (CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);.
```

Anschließend sollte die Software die Pixelwerte aus dem Array wieder auslesen und mit dem zweiten Filter falten.
Es verblieben jedoch horizontale Streifen auf dem Bild, die auch nach der Fehlersuche nicht erklärbar waren (Abbildung 9).

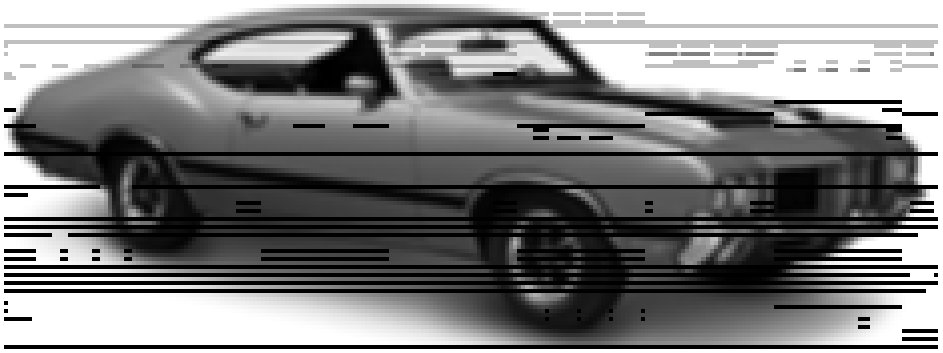

Abbildung 9: Gauß-Filter mit Streifen

Zusätzlich sollte die Größe des Arrays dynamisch zur Größe des eingelesenen Bildes angelegt werden.
Dazu legt das Programm die Variable `pic_size` an, die die Anzahl der Pixel des eingelesenen Bildes berechnet. Anschließend wird das Array `temp` der eben ermittelten Größe des Bildes anlegt und mithilfe der Funktion `clCreateBuffer` das Speicher-Objekt initialisiert. Dieses speichert die Daten des Arrays und wird vom Host zum Kernel geschickt. Folgende Anweisungen führen diese Prozedur aus.

```
int pic_size = width * height;
float temp [pic_size * 4];
cl_mem temp_buff = clCreateBuffer(context, CL_MEM_READ_WRITE |
                   CL_MEM_COPY_HOST_PTR, sizeof(float) *
                   pic_size, temp, &err);
```

Anschließend wird das Array mit

```
clSetKernelArg(sobel_grayscale, 2, sizeof(cl_mem), &temp_buff);
```

übergeben. Auf gleicher Art und Weise übergibt das Hostprogramm dem Kernel die Höhe und Breite des Bildes, welche für die Adressierung des Speicherplatzes im Arrayzwischenspeicher nötig sind.

Danach sollten noch zwei weitere Versionen geschrieben werden, die beide die gleiche Aufgabe mithilfe von zwei 1D-Filtern in zwei separaten Kernel verrichten.
In der ersten Variante sollten die Zwischenwerte in ein Array (Puffer-Objekt) geschrieben und in der Zweiten in ein Image-Objekt gespeichert werden. Dazu legt das Hostprogramm jeweils das entsprechende Speicher-Objekt an, welches dem ersten Kernel den Speicherplatz für die Zwischenwerte der Pixel als Argument übergibt. Der erste Kernel beschreibt diesen Speicherbereich mit den Zwischenwerten der Bildpunkte und gibt dem Host die Helligkeitswerte zurück. Dieser schickt dann die Zwischenwerte an den zweiten Kernel und startet selbigen. Am Ende des zweiten Kernels werden dann die bearbeiteten Pixelwerte in ein Image-Objekt geschrieben, dem Host übergeben und abschließend in ein PNG-File gespeichert.

Bei der zweiten Variante, in der ein Image-Objekt zu verwenden war, konnte in großen Teilen auf das Programm der ersten Version zurückgegriffen werden. Die Argumente der Abmessungen des Bildes entfallen für diese Variante und das Programm speichert die Pixelwerte in Image-, nicht in Puffer-Objekten.
Der Programmablaufplan der Version mit einem Puffer-Objekt ist in Abbildung 10 und der mit einem Image-Objekt in Abbildung 11 dargestellt.

Abbildung 10: Programmablaufplan Array-Zwischenspeicher

Abbildung 11: Programmablaufplan Image-Zwischenspeicher

Außerdem wurden für jede Version zwei *Offset*-Varianten erstellt, in denen der Kernel nur die Bildpunkte bearbeitet, deren Nachbarpixel vom Filter komplett einberechnet werden können. Das heißt, dass die Implementierung die äußersten Pixel nicht berücksichtigt, da der Filter sonst auf Pixel außerhalb des Bildes zugreift und das Ergebnis an den Rändern verfälscht.
In der ersten Variante werden die Pixel im Kernel überprüft und aussortiert. Das bedeutet, dass im Kernel jedes Pixel durch ein work-item existiert, aber gegebenenfalls bei der Berechnung der Farbwerte nicht berücksichtigt wird. Diese Version ist in Tabelle 2, 3 und 4 als “Kernel Offset“ aufgeführt.

Die zweite Version legt bereits im Hostprogramm die Größe des Bildes über die Argumente `global_work_offset` und `global_work_size` beim Aufruf des Kernels mit der Funktion `clEnqueueNDRangeKernel` fest. Damit werden die work-items für die Ränder des Bildes übergangen und im Kernel nicht durchlaufen. Diese Version ist in Tabelle 2, 3 und 4 als “Host Offset“ aufgeführt und Abbildung 12 zeigt ihren Programmablaufplan.

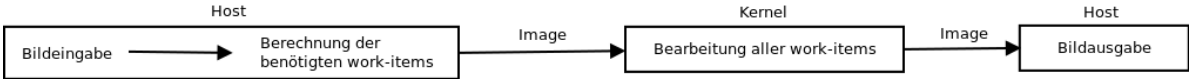

Abbildung 12: Programmablaufplan host offset

Zusätzlich zur Filterfunktion wurde in jedem Programm eine Laufzeitmessung der Kernel implementiert, welche zu einer Gegenüberstellung und Bewertung der Varianten dient. Diese beschränkt sich dabei nur auf die Laufzeit der Kernel von Anfang bis Ende der Ausführung von Kommandos auf dem Gerät (device).
Die Messung wird mithilfe der Funktion `event` realisiert, welche mit der Funktion `clEnqueueNDRangeKernel` assoziiert wird. Nach Fertigstellung der Berechnungen im Kernel schließt die Laufzeitmessung mit `clfinish(queue)` ab und kann anschließend ausgewertet werden. Dabei stehen verschiedene Optionen zur Wahl. Um die Vergleichbarkeit aller Programme zu wahren, wurde in jedem Programm dieser Abschlussarbeit die Laufzeit der Kernel identisch durchgeführt. Dabei ermittelt das Programm mit dem Argument `CL_PROFILING_COMMAND_START` in der Funktion `clGetEventProfilingInfo` die Zeit für den Anfang und mit dem Argument `CL_PROFILING_COMMAND_END` die Zeit für die Beendigung der Ausführung der Kommandos im Kernel. Durch Subtraktion der beiden gemessenen Zeiten wird anschließend die Laufzeit der Kernel errechnet.

Hinsichtlich der Laufzeit der Gaußversionen war zu erwarten, dass die Variante mit einem Kernel und zwei 1D-Filtern am schnellsten ist, da sie die geringste Anzahl an Rechenoperationen und nur zwei Datentransfers für die Zwischenspeicher besitzt (die Varianten mit zwei Kernels haben vier Datentransfers). Aufgrund der fehlerhaften Ausgabe des Bildes bei dieser Version, wurde aber auf die Aufführung in der Tabelle verzichtet. Bei den verbleibenden Versionen war zu erwarten, dass die Programme mit zwei 1D-Filtern und zwei Kernels schneller sind als die Versionen mit einem 2D-Filter, da weniger Rechenschritte für die Berechnung der Pixel benötigt werden. Dieser Effekt sollte besonders bei größeren Bildern zum Tragen kommen, da bei steigender Pixelanzahl die Rechenschritte des 2D-Filters schneller steigen. Zusätzlich war zu erwarten, dass das Programm mit dem Array als Zwischenspeicher langsamer ist als das mit dem Image als Zwischenspeicher. Ein Image-Objekt greift auf die Hardware der GPU zu, welche auf Bildverarbeitung ausgelegt ist, was dazu führt, dass die Pixelwerte sehr schnell ausgelesen werden können. Da der Standardfilter nur 3*3 Pixel groß ist, war ebenfalls zu erwarten, dass der Unterschied der Laufzeiten relativ gering ausfällt und nicht repräsentativ ist, da die 1D-Filter-Varianten ihren Geschwindigkeitsvorteil erst bei größeren Filtern oder Bildern signifikant entwickeln. Aufgrund dessen sind noch drei weitere Gauß-Filter mit unterschiedlicher Größe (5*5,

9*9 und 15*15 Pixel) in die Programme eingepflegt und zusätzlich jeweils ein kleineres (25*50 Pixel) sowie ein größeres Bild (263*498 Pixel) verwendet worden. Tabelle 1 zeigt die benötigten Rechenschritte für einen Bildpunkt während der Faltung für die jeweilige Filtergröße.

Filtergröße / Filtertyp	3x3	5x5	9x9	15x15	nxn
2x1D	6	10	18	30	2*n
1x2D	9	25	81	225	n^2

Tabelle 1: benötigte Rechenschritte pro Pixel

In der Tabelle ist zu erkennen, dass die Anzahl der Rechenschritte mit steigender Filtergröße bei 2x1D-Filtern linear und bei 2D-Filtern quadratisch zunimmt. Dies führt dazu, dass mit größer werdenden 1D-Filter mehr Laufzeit eingespart wird.

Die Laufzeitmessung wurde auf einem Laptop mit einer Nvidia GeForce 710M Grafikkarte durchgeführt. Diese besitzt zwei compute units und insgesamt 96 processing elements bei einer Speichertaktrate von 1800MHz und 775MHz Kerntakt.

Da die Laufzeiten speziell bei kleinen Filtern relativ stark schwanken, wurden die Programme 100 mal durchlaufen und die Durchschnittslaufzeit bestimmt. Dabei wurden folgende Ergebnisse ermittelt:

Laufzeiten				
Filtertyp	Filtergröße	kein Offset [µs]	Kernel Offset [µs]	Host Offset [µs]
2D, image	3x3	22,3	24,7	22,4
	5x5	24,1	26,2	24,2
	9x9	30,3	30,6	27,2
	15x15	46,5	39,2	33,2
1D, array	3x3	31,9	33,1	33,1
	5x5	33,1	33,2	34,7
	9x9	34,6	33,4	35,1
	15x15	35,5	34,5	36,3
1D, image	3x3	33,3	37,3	34,1
	5x5	36,2	39,7	34,7
	9x9	36,6	40,7	36,1
	15x15	37,7	41,2	38

Tabelle 2: Laufzeiten der Gauß-Varianten (Bildgröße 25*50 pixel)

Laufzeiten				
Filtertyp	Filtergröße	kein Offset [µs]	Kernel Offset [µs]	Host Offset [µs]
2D, image	3x3	43,5	53,9	38,3
	5x5	65,4	72,5	59,8
	9x9	166	155,6	144,3
	15x15	417	360,5	327,1
1D, array	3x3	67,5	68,6	54,3
	5x5	68,9	71,5	57,4
	9x9	77,4	77,9	65,9
	15x15	90,7	86,5	74
1D, image	3x3	67,2	74,3	62,1
	5x5	70,1	79,4	65,2
	9x9	80,8	84,5	69,1
	15x15	95,4	99,6	78,2

Tabelle 3: Laufzeiten der Gauß-Varianten (Bildgröße 90*233 pixel)

Laufzeiten				
Filtertyp	Filtergröße	kein Offset [µs]	Kernel Offset [µs]	Host Offset [µs]
2D, image	3x3	135,1	172,1	133,4
	5x5	296,1	293	282,5
	9x9	903,6	871,8	850,3
	15x15	2467,6	2317,7	2266,5
1D, array	3x3	194,6	205,7	195,5
	5x5	207,7	218,8	205,7
	9x9	260,2	279,9	254,5
	15x15	368,4	373,3	362,9
1D, image	3x3	215,5	225,3	215,9
	5x5	239,5	247,1	228,6
	9x9	264	318,5	260,7
	15x15	371,6	382	359,7

Tabelle 4: Laufzeiten der Gauß-Varianten (Bildgröße 263*498 pixel)

Die Laufzeittabellen zeigen, dass mit steigender Pixelzahl sowohl die Gesamtlaufzeit als auch die eingesparte Zeit der 1D-Versionen zunimmt. Ebenfalls ist zu erkennen, dass die 2D-Implementierung des Gauß-Filters bei kleinen Filter- und Bildgrößen durchschnittlich die schnellste ist. Erst mit steigender Pixelanzahl und Filtergröße sind die 1D-Versionen schneller und haben die Hypothek der vier Datentransfers durch ihren geringeren Rechenaufwand im jeweiligen Kernel wettgemacht. Damit wurde das erwartete Ergebnis aus der Vorbetrachtung bestätigt. Nur das Programm mit dem Image als Zwischenspeicher

war entgegen der vorherigen Annahme langsamer als das mit dem Array. Nur bei der Host-Offset-Version mit einem 15*15-Filter ist die Variante mit einer Textur als Zwischenspeicher schneller.

Der Quellcode der Version mit einem Kernel und 2D-Filter ist dem Dokument angehangen (Anlage 2).

5.3 SURF-Algorithmus

Anschließend wurde ein Teil des SURF-Algorithmus in OpenCL implementiert. SURF steht für “Speed-Up Robust Features“ und hat die Aufgabe Bildmerkmale (*blobs*) zu erkennen. [7]
Besondere Merkmale eines Bildes sind Muster, die sich von ihrer unmittelbaren Umgebung unterscheiden und mit Änderungen der Bildeigenschaften (z.B. Kontrast, Ecken und Kanten) einhergehen. Mithilfe dieser markanten Bildpunkte können dann Korrespondenzen mit Vergleichsbildern gefunden werden, die zum Detektieren von Objekten dienen.
Der SURF-Algorithmus kann beispielsweise zur Gesichtserkennung, Robotersteuerung und für das autonome Fahren verwendet werden. [8]

Die SURF-Methode besteht aus einer Reihe von Schritten, die nachfolgend erläutert werden.
Im ersten Schritt sieht der Algorithmus die Erstellung eines *Integralbilds* aus dem eingelesenen Bild vor, welches den Vorteil besitzt, beim Berechnen der Summe von Bildpunktintensitäten einer Fläche immer die gleiche Anzahl an Rechenschritten zu benötigen. Das bedeutet, dass die Laufzeit des Algorithmus unabhängig von der Größe des Bildes ist.

Im zweiten Schritt sucht der Algorithmus nach potenziellen *Feature-Punkten* mithilfe eines Hessematrix-basiertem Filters (SURFdetector). Es werden dabei blobs an Stellen detektiert, an denen die Determinante der Hessematrix ein Maximum hat. Die Hessematrix ist folgendermaßen definiert.

$$H(x,\sigma) = \begin{bmatrix} L_{xx}(x,\sigma) & L_{xy}(x,\sigma) \\ L_{xy}(x,\sigma) & L_{yy}(x,\sigma) \end{bmatrix} \tag{6}$$

Hierbei stellt $L_{xx}(x,\sigma)$ die Faltung der zweiten Ableitung des Gauß-Filters

$$\frac{\partial^2}{\partial x^2} g(\sigma) \tag{7}$$

mit dem Bild I im Punkt x dar.
Die Filtermasken der abgeleiteten Gauß-Filter sind in der Umsetzung ins Programm als Box-Filter ausgeführt. Abbildung 9 zeigt links die approximierten Filter D_{yy} und D_{xy} und

rechts die Gleichen als Box-Filter. Der nicht abgebildete dritte Filter D_{xx} entspricht dem um 90° gedrehten D_{yy}.

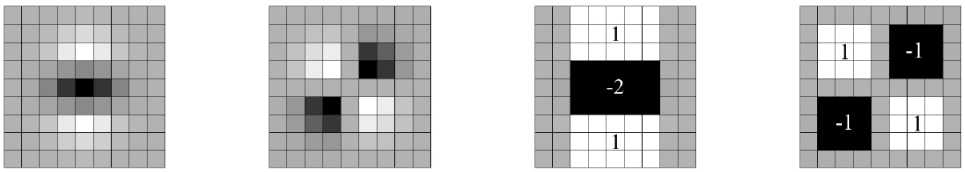

Abbildung 13: Box-Filter[7]

Nach den drei Faltungen errechnet der Algorithmus die Determinante der Hessematrix und damit den Feature-Wert des jeweiligen Bildpunktes mithilfe der folgenden Bildungsvorschrift.

$$det(H) = I_{xx} * I_{yy} - I_{xy}^2 \tag{8}$$

Die errechneten Feature-Punkte müssen bei verschiedenen Skalierungen gefunden werden, da die Suche von Korrespondenzen oft Vergleiche in Bildern anderer Größe stattfindet. Aus diesem Grund werden häufig die Originalbilder verkleinert und über Gauß-Filter weichgezeichnet. Um diesen Rechenaufwand zu umgehen, kann der Algorithmus mit aufskalierten Box-Filtern das Originalbild bearbeiten. Da zusätzlich mit einem Integralbild und mit einer GPU parallel gerechnet wird, ergibt sich kein Zeitunterschied bei den verschiedenen Filtern und eine schnelle Laufzeit. Diese Herangehensweise bringt des Weiteren den Vorteil mit sich, dem *aliasing* des Bildes beim Skalieren aus dem Weg zu gehen. Die Filtergröße startet in der ersten *Oktave* bei 9*9 Pixeln und vergrößert sich auf 15*15, 21*21 und 27*27. In einer Oktave ist der zu vergrößernde Wert immer doppelt so groß wie der der vorgehenden Oktave. Aufgrund der diskreten Filter hängt die Größe der aufeinanderfolgenden Skalierungen von der Länge l_0 ab, die einem Drittel der Filterbreite entspricht. Der erste Filter der ersten Oktave hat damit die Länge l_0=3 Pixel. Damit eine ungerade Kantenlänge des Filters und als Konsequenz daraus ein zentrales Pixel erhalten bleibt, muss l_0 um 2 Pixel erhöht werden. Daraus ergibt sich eine darauffolgende Filtergröße von 15*15 Pixeln. Bei der zweiten Oktave wird bei 15*15 Pixeln angefangen und l_0 um 2*2=4 Bildpunkte erhöht. Die zweite Stufe der zweiten Oktave besitzt demnach einen 27*27 Pixel großen Filter und ein l_0 von 9 Pixeln. Diese Prozedur kann beliebig fortgeführt werden. Allerdings fällt die Anzahl der detektierten Feature-Punkte je Oktave sehr schnell, was die Abbildung 14 illustriert.

[7] Quelle: https://www.academia.edu/36379818/Speeded-Up_Robust_Features_SURF_, S.348

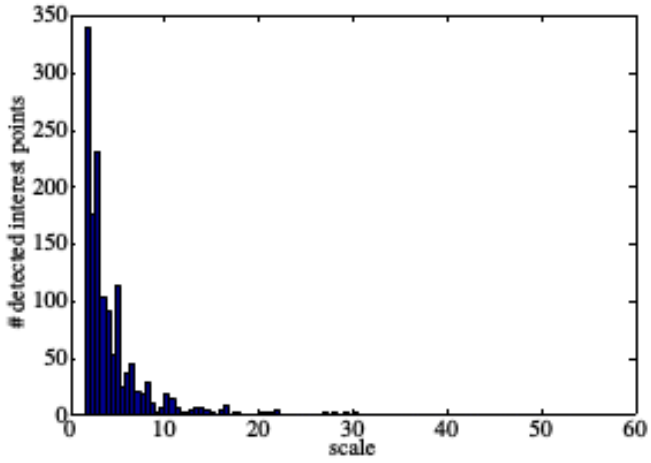

Abbildung 14: Feature-Detektierung[8]

Im darauffolgenden Schritt reduziert SURF die ermittelten Feature-Punkte in einer 3*3*3 Pixel Nachbarschaft mittels einer *Nicht-Maximum-Unterdrückung*, um sichere Punkte mit hohem Kontrast zu extrahieren. Von jedem verbliebenen Punkt werden der Pixelwert und dessen Position gespeichert.

Danach wird die Orientierung jedes Features mithilfe von Informationen aus der Nachbarschaft bestimmt, um *rotatorische Invarianz* des Bildes zu erzeugen. Dabei werden horizontale und vertikale Vektoren durch Filterung der Nachbarschaftspixel mit Haar-Wavelet-Filtern (horizontal und vertikal) errechnet und anschließend mit einem Gauß-Filter gewichtet. Diese Vektoren werden dann als Punkte im Raum entlang der Ordinate (horizontaler Vektor) und Abszisse (vertikaler Vektor) eingetragen und aufsummiert. Die dominante Orientierung wird durch Summation aller Ergebnisse in einem Fenster, welches sich durch den Raum verschiebt, abgeschätzt. Der längste Vektor der gesamten Fenster definiert letztlich die Richtung des Features (Abbildung 15).

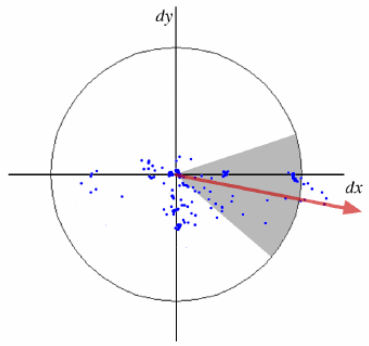

Abbildung 15: Feature-Orientierung[9]

[8]Quelle: https://www.academia.edu/36379818/Speeded-Up_Robust_Features_SURF_, S.351
[9]Quelle: https://www.academia.edu/36379818/Speeded-Up_Robust_Features_SURF_, S.352

Um den *Descriptor* zu bestimmen, wird im nächsten Schritt ein quadratisches Fenster um den Feature-Punkt angelegt und nach der im vorherigen Schritt ermittelten Orientierung ausgerichtet. Dieses Fenster wird dann in ein 4*4-Fenster unterteilt und mit dem Haar-Wavelet-Filter der Gradientenverlauf berechnet (Abbildung 16). Diese Descriptoreintragungen repräsentieren die zugrundeliegende Intensitätsstruktur der einzelnen Sub-Regionen.

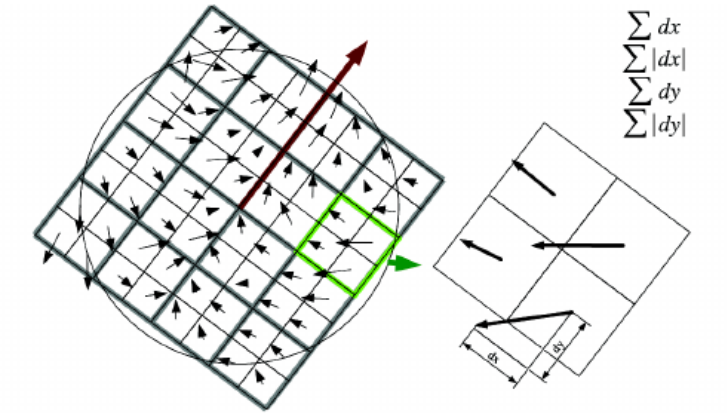

Abbildung 16: Descriptorbestimmung[10]

Im letzten Teil des Algorithmus (*matching stage*) wird das Vorzeichen des Laplace-Operators verwendet, um die Art des Kontrasts des Features (schwarz auf weiß oder umgekehrt) und somit die Übereinstimmung der Bilder zu bewerten. Die folgende Grafik zeigt den Programmablauf von SURF grafisch.

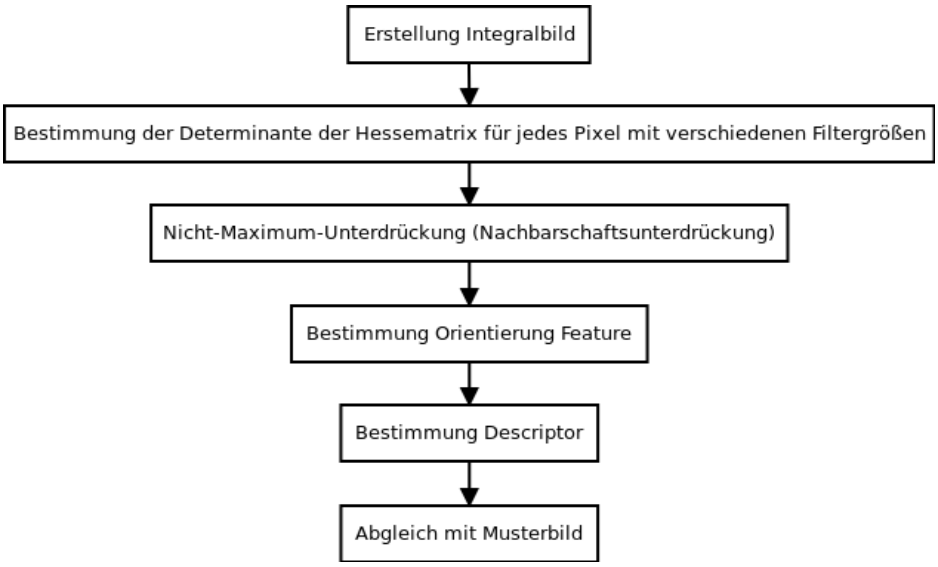

Abbildung 17: Programmablaufplan SURF-Algorithmus

[10]Quelle: https://www.academia.edu/36379818/Speeded-Up_Robust_Features_SURF_, S.352

Der zu implementierende Teil des SURF-Algorithmus sollte ohne die Erstellung eines Integralbildes die Feature-Punkte eines Bildes ermitteln. Das bedeutet, dass sowohl die Feature-Orientierung, als auch der Descriptor und die Matching-Stage nicht abzudecken war. Darüber hinaus sollte nur mit einer Filtergröße von 9*9 Pixeln gerechnet werden.

Zunächst wurde ein Kernel, mit der Aufgabe eine Faltung zu berechnen, erstellt. Dieser Kernel unterscheidet sich von den bisherigen Faltungskernels in der Hinsicht, dass der anzuwendende Filter als Array vom Host übergeben und nicht im Kernel deklariert wird. Zudem konnten im Kernel des Sobel-Filters bei der Berechnung der Faltung die benachbarten Pixel eines Bildpunktes aufgrund der kleinen Filtergröße direkt ausgelesen werden. Diese Herangehensweise ist bei einem 9*9-Filter mit seinen 81 Leseschritten jedoch nicht mehr praktikabel und schränkt die Übersichtlichkeit des Programmcodes weiter ein. Daher wurden die Lesebefehle mithilfe von zwei for-Schleifen und die Faltung mit einem Array programmiert. Da der Algorithmus drei Faltungen (in xx-, yy-, xy-Richtung) vorsieht, legt das Hostprogramm drei verschiedene Filter an und ruft den Faltungskernel dreimal nacheinander auf. Aufgrund der gleichen Rechenaufgaben bei den drei Faltungen, existiert der Kernel nur einmal in der Kerneldatei und der Host ruft ihn mit unterschiedlichen Argumenten (den drei Filtern) auf.

Anschließend wurde ein Kernel erstellt, der die Determinante der Hessematrix für jeden Bildpunkt bestimmt und als Textur zurückgibt. Dabei liest der Kernel die Helligkeitswerte der drei gefalteten und als Argument übergebenen Bilder und berechnet dann die Determinante mithilfe der folgenden Kommandozeile.

```
float4 pixel_D = pixel_Ixx * pixel_Iyy -
                 (pixel_Ixy * pixel_Ixy);
```

Die Determinantenwerte speichert der Kernel in einer Textur und gibt diese an den Host zurück.

Ein vierter Kernel sieht danach die Featureberechnung und *Nachbarschaftsunterdrückung* vor. Das lokale Maximum der Determinante stellt dabei das zu bestimmende Bildmerkmal dar. Der Host übergibt wieder die Determinantentextur und bildet für jedes Pixel den Betrag für die Featureberechnung. Hierfür wird auf die C-Funktion `fabs()` zurückgegriffen. Um die zu übertragende und weiter zu verarbeitende Datenmenge zu reduzieren, arbeitet der Kernel anschließend eine Nachbarschaftsunterdrückung ab. In diesem Abschnitt werden die Helligkeitswerte der acht Nachbarpixel des zu betrachtenden Bildpunktes ausgelesen und mit dem Wert des Mittelpunktes verglichen. Falls ein Wert der umliegenden Bildpunkte größer als der des zu filternden Pixels ist, stellt der betrachtete Bildpunkt kein lokales Maximum dar und wird daher nicht ins Ausgabebild geschrieben. Ein zusätzlicher Schwellwert stellt die Ausgabe ab einem bestimmten Helligkeitswert sicher, da Bildpunkte mit einem niedrigeren Helligkeitswert keinen signifikanten Feature-Punkt darstellen.
Im Programm wurde bei der Nachbarschaftsunterdrückung von allen umliegenden Bildpunkten das Maximum ermittelt und dann mit dem Wert des zu filternden Pixels vergli-

chen. Um ein Abschneiden (*clippen*) der Maximalwerte vor der Featureberechnung zu verhindern, werden die Extremwerte der drei Faltungen bestimmt und die errechneten Helligkeitswerte der einzelnen Bildpunkte mithilfe der folgenden Formel vor der Determinantenberechnung skaliert.

$$pixel\_value\_scaled = \frac{1.0f}{max\_value - min\_value} * (pixel\_value - min\_value) \quad (9)$$

Diese Skalierung wird im Kernel folgendermaßen umgesetzt.

```
float4 pixel_D = pixel_Ixx * pixel_Iyy -
                 (pixel_Ixy * pixel_Ixy);
```

Damit der Farbwert der Bildpunkte nach der Determinatenberechnung nicht noch einmal geclippt wird, speichert der Kernel das Ergebnis der Determinante in einem Array zwischen, sodass von den Faltungen am Anfang des Algorithmus bis zur Nachbarschaftsunterdrückung am Ende kein Wert abgeschnitten wird.

Zuletzt schreibt das Programm die gefilterten Bildpunkte in ein Image und sendet es dem Host, welcher das Bild in eine PNG-Datei schreibt und ausgibt. Die nachfolgende Abbildung zeigt den Programmablaufplan der Implementierung.

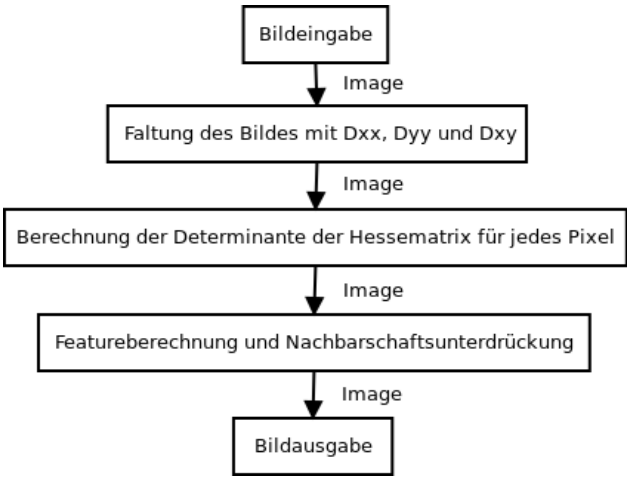

Abbildung 18: Programmablaufplan SURF-Implementierung

Das resultierende Bild ist in Abbildung 19 dargestellt.

Abbildung 19: Determinante in Array als Zwischenspeicher

Im Anschluss sollten die Programmdurchlaufzeiten reduziert werden.
Dabei wurden zunächst analog zum Gauß-Filter-Programm die 2D-Filter in jeweils einen horizontalen und vertikalen 1D-Filter umgerechnet und die Zwischenwerte in einem Array bzw. Image gespeichert. Anschließend wurde von jeder 1D-Filter-Version eine weitere Variante erstellt, bei der nicht drei Kernel nacheinander, sondern ein Kernel die drei Faltungen berechnet. Dabei wird die Eigenschaft des Datentyps float4, vier Kanäle zu besitzen, ausgenutzt. Der Kernel liest den Helligkeitswert des Bildpunktes vom unbearbeiteten Bild aus und speichert diesen in einem der vier Kanäle einer Variable vom Datentyp `float4` zwischen. Anschließend liest das Programm den Wert wieder aus der Variable aus und faltet diese mit den drei Filtern. Zum Schluss werden die drei Ergebnisse wieder in der gleichen `float4`-Variable gespeichert, in ein Array oder Image geschrieben und dem Host übergeben. Die folgende Grafik zeigt den Programmablauf der Variante mit zwei 1D-Filtern und einem Puffer-Objekt als Zwischenspeicher.

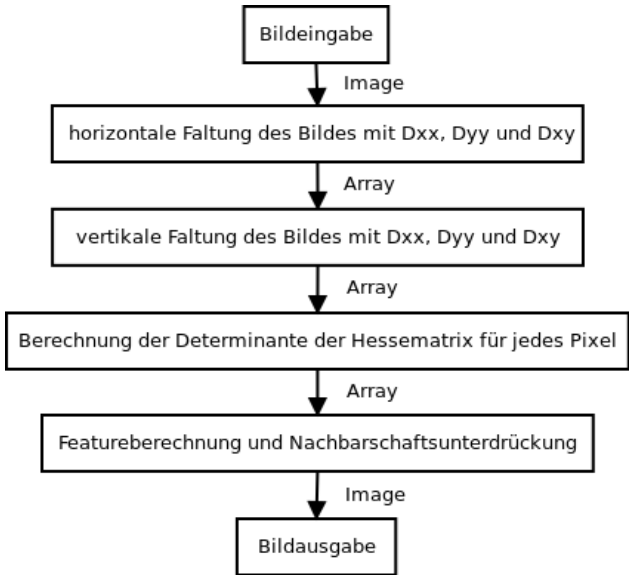

Abbildung 20: Programmablaufplan SURF mit 1D-Filtern und Array-Zwischenspeichern

Die folgenden Tabellen zeigen aufgrund der Laufzeitschwankungen wieder die Durchschnittslaufzeiten der verschiedenen Versionen nach 100 Durchläufen. Wie auch beim Gauß-Filter wurden verschieden große Bilder (25*50, 90*233 und 263*498 Pixel) verwendet. Es war zu erwarten, dass die Versionen mit einem Faltungskernel (one_convo_*) am schnellsten sind und die Varianten mit zwei 1D-Filtern schneller sind, als die mit einem 2D-Filter. Dieser Geschwindigkeitsvorteil sollte sich vor allem bei größeren Bildern bemerkbar machen. Analog zum Gauß-Filter war darüber hinaus zu erwarten, dass die Versionen mit einem Image als Zwischenspeicher (all_image) schneller sind als die mit einem Array (all_array). In der Variante “array_between“ werden die Zwischenwerte der Faltungen mit 1D-Filter in einem Array und das Endresultat in einer Textur gespeichert.

Filtertyp	all_image [µs]	array_between [µs]	all_array [µs]	offset [µs]
surf_2D	112,3	-	97,4	-
surf_1D	154,1	117,2	108,3	-
one_convo_2D	85,1	-	66,6	-
one_convo_1D	109,7	78,9	65,9	64,3

Tabelle 5: Laufzeiten der SURF-Varianten (Bildgröße 25*50 pixel)

Filtertyp	all_image [µs]	array_between [µs]	all_array [µs]	offset [µs]
surf_2D	540,9	-	560,8	-
surf_1D	306,5	283,2	301	-
one_convo_2D	274,4	-	259,3	-
one_convo_1D	257	209,8	206,5	207,2

Tabelle 6: Laufzeiten der SURF-Varianten (Bildgröße 90*233 pixel)

Filtertyp	all_image [µs]	array_between [µs]	all_array [µs]	offset [µs]
surf_2D	3013,6	-	3221,4	-
surf_1D	1222,9	1259,1	1476,4	-
one_convo_2D	1346,4	-	1439,4	-
one_convo_1D	1047,3	929,5	1120,7	1123,6

Tabelle 7: Laufzeiten der SURF-Varianten (Bildgröße 263*498 pixel)

Die Tabellen zeigen, dass die Versionen mit einem Faltungskernel schneller sind, als die, bei denen der Faltungskernel mehrmals aufgerufen wird. Des Weiteren ist zu erkennen, dass mit steigender Bildgröße die Gesamtlaufzeiten steigen und die eingesparte Zeit im Verhältnis wächst. So brauchte die all_image-Version von one_convo_1D bei der Bildgröße 263*498 Pixel ungefähr nur ein Drittel der Zeit von surf_2D. Bei der Messung mit einem Bild der Größe 90*233 Pixel brauchte die Version one_convo_1D hingegen nur etwas weniger als die Hälfte der Zeit. Darüber hinaus sind die Varianten mit zwei 1D-Filtern schneller, als die mit einem 2D-Filter. Diese Annahmen wurden somit bestätigt.

Entgegen der Annahme, dass die Varianten mit Arrays als Zwischenspeicher langsamer sind als die mit Texturen, ist für kleine Bilder die schnellste Version des SURF-Algorithmus die mit zwei 1D-Filtern und einem Array als Zwischenspeicher. Erst bei größeren Bildern weisen die Versionen mit Texturen als Zwischenspeicher wieder die besseren Laufzeiten auf. Speziell in Tabelle 7 ist dies zu erkennen. Damit ist festzuhalten, dass die Texturzwischenspeicher erst dann die Laufzeit reduzieren, wenn der Kernel durch die Abfrage der Helligkeitswerte von den Bildpunkten oft auf das Image zugreift. Zusätzlich zu den bereits erwähnten Versionen, wurde analog zum Gauß-Filter für die schnellste Version in der Messung mit einer Bildgröße von 90*233 Pixeln noch eine mit Offset erstellt. Diese ist nahezu gleich schnell, liefert jedoch zuverlässigere und damit bessere Ergebnisse.

Der Quellcode des Kernels der Version one_convo_1D all_array ist dem Dokument angehangen (Anlage 3).

6 Fazit der Abschlussarbeit

Im Rahmen der Abschlussarbeit gelang es Filter wie Sobel und Gauß zu implementieren und lauffähig zu gestalten. Zusätzlich konnten verschiedene Wege, ob konventioneller Natur (mit Texturen), oder auf alternativer Art (mit Arrays), eingeschlagen werden, die zum gleichen Ziel mit unterschiedlichen Laufzeiten führten. Darüber hinaus gelang es einen Großteil des SURF-Algorithmus zu implementieren und auch hier die Laufzeiten miteinander zu vergleichen. Des Weiteren konnten die Laufzeiten in allen Varianten des Gauß-Filters und des SURF-Algorithmus verbessert und die verschiedenen Varianten verglichen werden. In einigen Fällen wurde die Laufzeit sogar auf ein Drittel reduziert. Ein OpenCL-fähiges Image auf ein Wandboard mit einem i.MX6 Quad Prozessor zu spielen, um die geschriebenen Programme auf einem Eingebetteten System laufen zu lassen und die Laufzeiten zu bestimmen, gelang hingegen nicht. Folglich konnten die Laufzeiten vom Laptop nicht mit denen des embedded-systems verglichen werden.

7 Ausblick

Der bereits überarbeitete SURF-Algorithmus hat noch Verbesserungspotenzial, bezüglich seiner Laufzeit. Beispielsweise werden aktuell die Werte der nachbarschaftsunterdrückten Pixel in ein Array geschrieben, was der Größe des eingelesenen Bildes entspricht. Hier könnten mithilfe einer Atomic-Funktion nur so viele Werte zurückgelesen werden, wie hineingeschrieben wurden. Zusätzlich könnte man im Hostprogramm die Kernelgröße auf die optimale work-group-Größe einstellen, was die Ressourcen der Hardware besser ausnutzen würde. Beides würde dazu führen, dass noch weniger Rechenaufwand betrieben wird und die Laufzeit sich verkürzt.

8 Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen Hilfsmittel als angegeben verwendet habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Ort: Chemnitz

Datum: 27. September 2019 Unterschrift:

9 Quellen

[1] https://www.khronos.org/opencl/ (19.06.2019)

[2] Matthew Scarpino: OpenCL in Action, Manning, 2011, S.6-7

[3] Matthew Scarpino: OpenCL in Action, Manning, 2011, S.88

[4] https://de.wikipedia.org/wiki/Sobel-Operator (19.06.2019)

[5] Aaftab Munshi, Benedict R. Gaster, Timothy G. Mattson, James Fung, Dan Ginsburg: OpenCL Programming Guide, Addison-Wesley, S.410

[6] https://de.wikipedia.org/wiki/Gau%C3%9F-Filter (19.06.2019)

[7] https://www.academia.edu /36379818/Speeded-Up_Robust_Features_SURF_ (19.06.2019)

[8] https://www.informatik.hu-berlin.de/ de/forschung/gebiete/viscom/teaching/media/cphoto10/cphoto10_04.pdf (19.06.2019)

10 Anlagen

Anlage 1: Quellcode Sobel-Filter

Hostprogramm

```
//#define _CRT_SECURE_NO_WARNING
#define PROGRAM_FILE "sobel_filter_oclpg.cl"

#define PNG_DEBUG 3
#include <png.h>

#define INPUT_FILE "input_8bit.png"
//#define INPUT_FILE "4.png"
#define OUTPUT_FILE "output.png"
#define OUTPUT_FILE_UNTOUCHED "output_untouched.png"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <CL/cl.h>

void read_image_data(const char* filename, png_bytep* data, size_t* w,
    size_t* h) {

   int i;

   /* Open input file */
   FILE *png_input;
   if((png_input = fopen(filename, "rb")) == NULL) {
      perror("Can't read input image file");
      exit(1);
   }

   /* Read image data */
   png_structp png_ptr = png_create_read_struct(PNG_LIBPNG_VER_STRING,
       NULL, NULL, NULL);
   png_infop info_ptr = png_create_info_struct(png_ptr);
   png_init_io(png_ptr, png_input);
   png_read_info(png_ptr, info_ptr);
   *w = png_get_image_width(png_ptr, info_ptr);
   *h = png_get_image_height(png_ptr, info_ptr);

   /* Allocate memory and read image data */
   *data = malloc(*h * png_get_rowbytes(png_ptr, info_ptr));
   for(i=0; i<*h; i++) {
      png_read_row(png_ptr, *data + i * png_get_rowbytes(png_ptr, info_ptr
          ), NULL);
   }

   /* Close input file */
```

```
   png_read_end(png_ptr, info_ptr);
   png_destroy_read_struct(&png_ptr, &info_ptr, (png_infopp)NULL);
   fclose(png_input);
}
}
void write_image_data(const char* filename, png_bytep data, size_t w,
    size_t h) {

   int i;

   /* Open output file */
   FILE *png_output;
   if((png_output = fopen(filename, "wb")) == NULL) {
      perror("Create␣output␣image␣file");
      exit(1);
   }

   /* Write image data */
   png_structp png_ptr = png_create_write_struct(PNG_LIBPNG_VER_STRING,
       NULL, NULL, NULL);
   png_infop info_ptr = png_create_info_struct(png_ptr);
   png_init_io(png_ptr, png_output);
   png_set_IHDR(png_ptr, info_ptr, w, h, 8,
         PNG_COLOR_TYPE_GRAY,
         PNG_INTERLACE_NONE,
         PNG_COMPRESSION_TYPE_BASE, PNG_FILTER_TYPE_BASE);
   png_write_info(png_ptr, info_ptr);
   for(i=0; i<h; i++) {
      png_write_row(png_ptr, data + i*png_get_rowbytes(png_ptr, info_ptr))
          ;
   }

   /* Close file */
   png_write_end(png_ptr, NULL);
   png_destroy_write_struct(&png_ptr, &info_ptr);
   fclose(png_output);
}

int main(int argc, char **argv) {

   /* Host/device data structures */
   cl_device_id device;
   cl_platform_id platform;
   cl_context context;
   cl_command_queue queue;
   cl_event event;
   cl_program program;
   cl_kernel sobel_grayscale;
   cl_int err;
   size_t global_size[2];

   /* Image data */
```

```
png_bytep pixels_in, pixels_out;
cl_image_format png_format_gray, png_format_rgba;
cl_mem input_image, output_image;
cl_image_desc desc;
size_t origin[3], region[3];
size_t width, height;

cl_ulong time_start, time_end;
double total_time;

/* Open input file and read image data */
read_image_data(INPUT_FILE, &pixels_in, &width, &height);

printf("height:%zu\nwidth:%zu\n",height,width);

write_image_data("output_untouched.png", pixels_in, width, height);

/* Identify a platform */
err = clGetPlatformIDs(1, &platform, NULL);
if(err < 0) {
   perror("Couldn't identify a platform");
   exit(1);
}

/* Access a device */
err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
if(err == CL_DEVICE_NOT_FOUND) {
   err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1, &device, NULL)
      ;
}
if(err < 0) {
   perror("Couldn't access any devices");
   exit(1);
}

/* Create a context */
context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);
if(err < 0) {
   perror("Couldn't create a context");
   exit(1);
}

/* Build the program from file and compile it*/
FILE *program_handle;
char *program_buffer, *program_log;
size_t program_size, log_size;
//int err;

/* Read program file and place content into buffer */
program_handle = fopen(PROGRAM_FILE, "r");
if(program_handle == NULL) {
   perror("Couldn't find the program file");
```

```
   exit(1);
}
fseek(program_handle, 0, SEEK_END);
program_size = ftell(program_handle);
rewind(program_handle);
program_buffer = (char*)malloc(program_size + 1);
program_buffer[program_size] = '\0';
fread(program_buffer, sizeof(char), program_size, program_handle);
fclose(program_handle);

/* Create program from file */
program = clCreateProgramWithSource(context, 1,
   (const char**)&program_buffer, &program_size, &err);
if(err < 0) {
   perror("Couldn't create the program");
   exit(1);
}
free(program_buffer);

err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if(err < 0) {

   /* Find size of log and print to std output */
   clGetProgramBuildInfo(program, device,
            CL_PROGRAM_BUILD_LOG, 0, NULL, &log_size);
   program_log = (char*) malloc(log_size + 1);
   program_log[log_size] = '\0';
   clGetProgramBuildInfo(program, device,
            CL_PROGRAM_BUILD_LOG, log_size + 1, program_log,
            NULL);
   printf("%s\n", program_log);
   //free(program_log);
   //printf("%d", log_size);

   exit(1);
}


/* Create a kernel */
sobel_grayscale = clCreateKernel(program, "sobel_grayscale", &err);
if(err < 0) {
  printf("Couldn't create  kernel: %d", err);
  exit(1);
}


/* Create image object */
png_format_gray.image_channel_order = CL_LUMINANCE;
png_format_gray.image_channel_data_type = CL_UNORM_INT8;
//png_format_rgba.image_channel_order = CL_RGBA;
//png_format_rgba.image_channel_data_type = CL_UNSIGNED_INT8;
```

```
desc.image_type = CL_MEM_OBJECT_IMAGE2D;
desc.image_width = width;
desc.image_height = height;
desc.image_depth = 0;
desc.image_array_size = 0;
desc.image_row_pitch = 0;
desc.image_slice_pitch = 0;
desc.num_mip_levels = 0;
desc.num_samples = 0;
desc.buffer = NULL;

input_image = clCreateImage(context,
             CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
             &png_format_gray, &desc, (void*)pixels_in, &err);

output_image = clCreateImage(context,
             CL_MEM_WRITE_ONLY, &png_format_gray, &desc, NULL, &err);

/* Create kernel arguments */
err = clSetKernelArg(sobel_grayscale, 0, sizeof(cl_mem), &input_image);
err |= clSetKernelArg(sobel_grayscale, 1, sizeof(cl_mem), &output_image
    );
if(err < 0) {
   printf("Couldn't set a kernel(hor) argument");
   exit(1);
};

/* Create a command queue */
queue = clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE
    , &err);
if(err < 0) {
   perror("Couldn't create a command queue");
   exit(1);
};

/* Enqueue kernel */
global_size[0] = width; global_size[1] = height;
err = clEnqueueNDRangeKernel(queue, sobel_grayscale, 2, NULL,
                             global_size, NULL, 0, NULL,  &event);
if(err < 0) {
       printf("error: %i",err);
   perror("Couldn't enqueue the kernel(hor)");
   exit(1);
}

//Profiling
clFinish(queue);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(
    time_start), &time_start, NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(
    time_end), &time_end, NULL);
total_time = time_end-time_start;
```

```
    printf("Kernel time is: %0.3f us\n", total_time/1000.0);

    /* Read the image object */
    pixels_out = malloc(height*width*32);
    origin[0] = 0; origin[1] = 0; origin[2] = 0;
    region[0] = width; region[1] = height; region[2] = 1;
    err = clEnqueueReadImage(queue, output_image, CL_TRUE, origin,
          region, 0, 0, (void*)pixels_out, 0, NULL, NULL);
    if(err < 0) {
           printf("errorcode:%i\n",err);
       perror("Couldn't read from the image object");
       exit(1);
    }


    write_image_data(OUTPUT_FILE, pixels_out, width, height);


    /* Deallocate resources */
    free(pixels_in);
    free(pixels_out);
    clReleaseMemObject(input_image);
    clReleaseMemObject(output_image);
    clReleaseKernel(sobel_grayscale);
    clReleaseCommandQueue(queue);
    clReleaseProgram(program);
    clReleaseContext(context);
    return 0;
}
```

Kernelprogramm

```
__constant sampler_t sampler =  CLK_NORMALIZED_COORDS_TRUE |
                                CLK_ADDRESS_CLAMP_TO_EDGE |
                                CLK_FILTER_LINEAR;

__kernel void sobel_grayscale(read_only image2d_t src, write_only
    image2d_t dst)
{
   int x = (int)get_global_id(0);
   int y = (int)get_global_id(1);

   float4 p00 = read_imagef(src, sampler, (int2)(x - 1, y - 1));
   float4 p10 = read_imagef(src, sampler, (int2)(x, y - 1));
   float4 p20 = read_imagef(src, sampler, (int2)(x + 1, y - 1));

   float4 p01 = read_imagef(src, sampler, (int2)(x - 1, y));
   float4 p21 = read_imagef(src, sampler, (int2)(x + 1, y));

   float4 p02 = read_imagef(src, sampler, (int2)(x - 1, y + 1));
   float4 p12 = read_imagef(src, sampler, (int2)(x, y + 1));
   float4 p22 = read_imagef(src, sampler, (int2)(x + 1, y + 1));

   float gx = -p00.x + p20.x + 2.0f * (p21.x - p01.x) - p02.x + p22.x;

   float gy = -p00.x - p20.x + 2.0f * (p12.x - p10.x) + p02.x + p22.x;

   float g = native_sqrt(gx * gx + gy * gy);

   write_imagef(dst, (int2)(x, y), (float4)(g, 1.0f, 1.0f, 1.0f));
}
```

Anlage 2: Quellcode Kernel 2D-Gauß-Filter

```
__constant sampler_t sampler =  CLK_NORMALIZED_COORDS_TRUE |
                                CLK_ADDRESS_CLAMP_TO_EDGE |
                                CLK_FILTER_LINEAR;

   #define FILTER_SIZE 3*3
   #define FILTER_HALF_SIZE (FILTER_SIZE-1)/2
   #define FILTER_WIDTH 3
   #define FILTER_HALF_WIDTH (FILTER_WIDTH-1)/2

__kernel void sobel_grayscale (read_only image2d_t src, write_only
    image2d_t dst)
{
   int x = (int)get_global_id(0);
   int y = (int)get_global_id(1);
   float4 output = (float4)(0.0f, 0.0f, 0.0f, 0.0f);

   #if FILTER_SIZE == 3*3
   float filter[FILTER_SIZE] = {1.0f/16.0f, 2.0f/16.0f, 1.0f/16.0f,
                                2.0f/16.0f, 4.0f/16.0f, 2.0f/16.0f,
                                1.0f/16.0f, 2.0f/16.0f, 1.0f/16.0f};
   #endif

   #if FILTER_SIZE == 5*5
   float filter[FILTER_SIZE] = {1.0f/100.0f, 2.0f/100.0f,   4.0f/100.0f,
      2.0f/100.0f, 1.0f/100.0f,
                                2.0f/100.0f, 4.0f/100.0f,   8.0f/100.0f,
                                   4.0f/100.0f, 2.0f/100.0f,
                                4.0f/100.0f, 8.0f/100.0f,  16.0f/100.0f,
                                   8.0f/100.0f, 4.0f/100.0f,
                                2.0f/100.0f, 4.0f/100.0f,   8.0f/100.0f,
                                   4.0f/100.0f, 2.0f/100.0f,
                                1.0f/100.0f, 2.0f/100.0f,   4.0f/100.0f,
                                   2.0f/100.0f, 1.0f/100.0f};
   #endif

   #if FILTER_SIZE == 9*9
   float filter[FILTER_SIZE] = {1.0f/2116.0f,  2.0f/2116.0f,  4.0f/2116.0f
      ,   8.0f/2116.0f,  16.0f/2116.0f,   8.0f/2116.0f,  4.0f/2116.0f,
      2.0f/2116.0f,  1.0f/2116.0f,
                                2.0f/2116.0f,  4.0f/2116.0f,  8.0f/2116.0f
                                   ,  16.0f/2116.0f,  32.0f/2116.0f,  16.0
                                   f/2116.0f,  8.0f/2116.0f,  4.0f/2116.0f
                                   ,  2.0f/2116.0f,
                                4.0f/2116.0f,  8.0f/2116.0f, 16.0f/2116.0f
                                   ,  32.0f/2116.0f,  64.0f/2116.0f,  32.0
                                   f/2116.0f, 16.0f/2116.0f,  8.0f/2116.0f
                                   ,  4.0f/2116.0f,
                                8.0f/2116.0f, 16.0f/2116.0f, 32.0f/2116.0f
                                   ,  64.0f/2116.0f, 128.0f/2116.0f,  64.0
                                   f/2116.0f, 32.0f/2116.0f, 16.0f/2116.0f
```

```
                                    ,  8.0f/2116.0f,
                                16.0f/2116.0f, 32.0f/2116.0f, 64.0f/2116.0f
                                   , 128.0f/2116.0f, 256.0f/2116.0f, 128.0f
                                   /2116.0f, 64.0f/2116.0f, 32.0f/2116.0f,
                                   16.0f/2116.0f,
                                 8.0f/2116.0f, 16.0f/2116.0f, 32.0f/2116.0f
                                    ,  64.0f/2116.0f, 128.0f/2116.0f,  64.0
                                    f/2116.0f, 32.0f/2116.0f, 16.0f/2116.0f
                                    ,  8.0f/2116.0f,
                                 4.0f/2116.0f,  8.0f/2116.0f, 16.0f/2116.0f
                                    ,  32.0f/2116.0f,  64.0f/2116.0f,  32.0
                                    f/2116.0f, 16.0f/2116.0f,  8.0f/2116.0f
                                    ,  4.0f/2116.0f,
                                 2.0f/2116.0f,  4.0f/2116.0f,  8.0f/2116.0f
                                    ,  16.0f/2116.0f,  32.0f/2116.0f,  16.0
                                    f/2116.0f,  8.0f/2116.0f,  4.0f/2116.0f
                                    ,  2.0f/2116.0f,
                                 1.0f/2116.0f,  2.0f/2116.0f,  4.0f/2116.0f
                                    ,   8.0f/2116.0f,  16.0f/2116.0f,   8.0
                                    f/2116.0f,  4.0f/2116.0f,  2.0f/2116.0f
                                    ,  1.0f/2116.0f};
#endif

#if FILTER_SIZE == 15*15
float filter[FILTER_SIZE] = {1.0f/145924.0f,   2.0f/145924.0f,   4.0f
   /145924.0f,    8.0f/145924.0f,   16.0f/145924.0f,   32.0f/145924.0f,
      64.0f/145924.0f,   128.0f/145924.0f,   64.0f/145924.0f,   32.0f
   /145924.0f,   16.0f/145924.0f,    8.0f/145924.0f,   4.0f/145924.0f,
     2.0f/145924.0f,   1.0f/145924.0f,
                              2.0f/145924.0f,   4.0f/145924.0f,   8.0f
                                 /145924.0f,   16.0f/145924.0f,   32.0f
                                 /145924.0f,   64.0f/145924.0f,  128.0f
                                 /145924.0f,   256.0f/145924.0f,  128.0f
                                 /145924.0f,   64.0f/145924.0f,   32.0f
                                 /145924.0f,   16.0f/145924.0f,   8.0f
                                 /145924.0f,   4.0f/145924.0f,   2.0f
                                 /145924.0f,
                              4.0f/145924.0f,   8.0f/145924.0f,  16.0f
                                 /145924.0f,   32.0f/145924.0f,   64.0f
                                 /145924.0f,  128.0f/145924.0f,  256.0f
                                 /145924.0f,   512.0f/145924.0f,  256.0f
                                 /145924.0f,  128.0f/145924.0f,   64.0f
                                 /145924.0f,   32.0f/145924.0f,  16.0f
                                 /145924.0f,   8.0f/145924.0f,   4.0f
                                 /145924.0f,
                              8.0f/145924.0f,  16.0f/145924.0f,  32.0f
                                 /145924.0f,   64.0f/145924.0f,  128.0f
                                 /145924.0f,  256.0f/145924.0f,  512.0f
                                 /145924.0f,  1024.0f/145924.0f,  512.0f
                                 /145924.0f,  256.0f/145924.0f,  128.0f
                                 /145924.0f,   64.0f/145924.0f,  32.0f
                                 /145924.0f,  16.0f/145924.0f,   8.0f
```

```
     /145924.0f,
   16.0f/145924.0f,  32.0f/145924.0f,  64.0f
      /145924.0f,  128.0f/145924.0f,  256.0f
      /145924.0f,  512.0f/145924.0f, 1024.0f
      /145924.0f,  2048.0f/145924.0f, 1024.0f
      /145924.0f,  512.0f/145924.0f,  256.0f
      /145924.0f,  128.0f/145924.0f,  64.0f
      /145924.0f,  32.0f/145924.0f,  16.0f
      /145924.0f,
   32.0f/145924.0f,  64.0f/145924.0f, 128.0f
      /145924.0f,  256.0f/145924.0f,  512.0f
      /145924.0f, 1024.0f/145924.0f, 2048.0f
      /145924.0f,  4096.0f/145924.0f, 2048.0f
      /145924.0f, 1024.0f/145924.0f,  512.0f
      /145924.0f,  256.0f/145924.0f, 128.0f
      /145924.0f,  64.0f/145924.0f,  32.0f
      /145924.0f,
   64.0f/145924.0f, 128.0f/145924.0f, 256.0f
      /145924.0f,  512.0f/145924.0f, 1024.0f
      /145924.0f, 2048.0f/145924.0f, 4096.0f
      /145924.0f,  8192.0f/145924.0f, 4096.0f
      /145924.0f, 2048.0f/145924.0f, 1024.0f
      /145924.0f,  512.0f/145924.0f, 256.0f
      /145924.0f, 128.0f/145924.0f,  64.0f
      /145924.0f,
  128.0f/145924.0f, 256.0f/145924.0f, 512.0f
     /145924.0f, 1024.0f/145924.0f, 2048.0f
     /145924.0f, 4096.0f/145924.0f, 8192.0f
     /145924.0f, 16384.0f/145924.0f, 8192.0f
     /145924.0f, 4096.0f/145924.0f, 2048.0f
     /145924.0f, 1024.0f/145924.0f, 512.0f
     /145924.0f, 256.0f/145924.0f, 128.0f
     /145924.0f,
   64.0f/145924.0f, 128.0f/145924.0f, 256.0f
      /145924.0f,  512.0f/145924.0f, 1024.0f
      /145924.0f, 2048.0f/145924.0f, 4096.0f
      /145924.0f,  8192.0f/145924.0f, 4096.0f
      /145924.0f, 2048.0f/145924.0f, 1024.0f
      /145924.0f,  512.0f/145924.0f, 256.0f
      /145924.0f, 128.0f/145924.0f,  64.0f
      /145924.0f,
   32.0f/145924.0f,  64.0f/145924.0f, 128.0f
      /145924.0f,  256.0f/145924.0f,  512.0f
      /145924.0f, 1024.0f/145924.0f, 2048.0f
      /145924.0f,  4096.0f/145924.0f, 2048.0f
      /145924.0f, 1024.0f/145924.0f,  512.0f
      /145924.0f,  256.0f/145924.0f, 128.0f
      /145924.0f,  64.0f/145924.0f,  32.0f
      /145924.0f,
   16.0f/145924.0f,  32.0f/145924.0f,  64.0f
      /145924.0f,  128.0f/145924.0f,  256.0f
      /145924.0f,  512.0f/145924.0f, 1024.0f
```

```
                                    /145924.0f,  2048.0f/145924.0f, 1024.0f
                                    /145924.0f,  512.0f/145924.0f,  256.0f
                                    /145924.0f,  128.0f/145924.0f,  64.0f
                                    /145924.0f,  32.0f/145924.0f,  16.0f
                                    /145924.0f,
                                  8.0f/145924.0f,  16.0f/145924.0f,  32.0f
                                     /145924.0f,   64.0f/145924.0f,  128.0f
                                     /145924.0f,  256.0f/145924.0f,  512.0f
                                     /145924.0f,  1024.0f/145924.0f,  512.0f
                                     /145924.0f,  256.0f/145924.0f,  128.0f
                                     /145924.0f,   64.0f/145924.0f,  32.0f
                                     /145924.0f,  16.0f/145924.0f,   8.0f
                                     /145924.0f,
                                  4.0f/145924.0f,   8.0f/145924.0f,  16.0f
                                     /145924.0f,   32.0f/145924.0f,   64.0f
                                     /145924.0f,  128.0f/145924.0f,  256.0f
                                     /145924.0f,   512.0f/145924.0f,  256.0f
                                     /145924.0f,  128.0f/145924.0f,   64.0f
                                     /145924.0f,   32.0f/145924.0f,  16.0f
                                     /145924.0f,   8.0f/145924.0f,   4.0f
                                     /145924.0f,
                                  2.0f/145924.0f,   4.0f/145924.0f,   8.0f
                                     /145924.0f,   16.0f/145924.0f,   32.0f
                                     /145924.0f,   64.0f/145924.0f,  128.0f
                                     /145924.0f,   256.0f/145924.0f,  128.0f
                                     /145924.0f,   64.0f/145924.0f,   32.0f
                                     /145924.0f,   16.0f/145924.0f,   8.0f
                                     /145924.0f,   4.0f/145924.0f,   2.0f
                                     /145924.0f,
                                  1.0f/145924.0f,   2.0f/145924.0f,   4.0f
                                     /145924.0f,    8.0f/145924.0f,   16.0f
                                     /145924.0f,   32.0f/145924.0f,   64.0f
                                     /145924.0f,   128.0f/145924.0f,   64.0f
                                     /145924.0f,   32.0f/145924.0f,   16.0f
                                     /145924.0f,    8.0f/145924.0f,   4.0f
                                     /145924.0f,   2.0f/145924.0f,   1.0f
                                     /145924.0f};
    #endif

    int filter_index = 0;
    for (int i=-FILTER_HALF_WIDTH; i<=FILTER_HALF_WIDTH; i++) {
        for (int j=-FILTER_HALF_WIDTH; j<=FILTER_HALF_WIDTH; j++) {
             float4 pixel = read_imagef(src, sampler, (int2)(x,y) + (int2)(
                j,i));
             output.x += pixel.x*filter[filter_index++];
        }
    }

    write_imagef(dst, (int2)(x, y), (float4)(output.x, 1.0f, 1.0f, 1.0f));
}
```

Anlage 3: Quellcode Kernel SURF-Filter Version one_convo_1D all_array

```
//all array
//1D-Filter

//#pragma OPENCL EXTENSION cl_khr_global_int32_base_atomics : enable

//#define FILTER_SIZE 9
//#define FILTER_HALF_SIZE (FILTER_SIZE-1)/2
#define FILTER_WIDTH 9
#define FILTER_HALF_WIDTH (FILTER_WIDTH-1)/2

#define TOP_LEFT [x + y* width - width -1]
#define TOP_CENTER [x + y* width - width]
#define TOP_RIGHT [x + y* width - width +1]
#define LEFT [x + y* width -1]
#define RIGHT [x + y* width +1]
#define LOWER_LEFT [x + y* width + width -1]
#define LOWER_CENTER [x + y* width + width]
#define LOWER_RIGHT [x + y* width + width +1]

__constant sampler_t sampler =  CLK_NORMALIZED_COORDS_TRUE |
                                CLK_ADDRESS_CLAMP_TO_EDGE |
                                CLK_FILTER_LINEAR;

__kernel void convolution_horizontal (read_only image2d_t src,
                                      __constant int* dim,
                                      __constant float* filter_Lxx,
                                      __constant float* filter_Lyy,
                                      __constant float* filter_Lxy,
                                      __global float* array_value_Ixx,
                                      __global float* array_value_Iyy,
                                      __global float* array_value_Ixy)
{
   int x = (int)get_global_id(0);
   int y = (int)get_global_id(1);
   int width = dim[0];
   int height = dim[1];
   float4 output = (float4)(0.0f, 0.0f, 0.0f, 0.0f);
   array_value_Ixx [x + y* width] = 0.0f;
   array_value_Iyy [x + y* width] = 0.0f;
   array_value_Ixy [x + y* width] = 0.0f;

   if(y>=FILTER_HALF_WIDTH && y<(height-FILTER_HALF_WIDTH)){
   int filter_index = 0;
   for (int i=-FILTER_HALF_WIDTH; i<=FILTER_HALF_WIDTH; i++) {
        float4 pixel = read_imagef(src, sampler, (int2)(x,y) + (int2)(i,0)
           );
        output.x += pixel.x*filter_Lxx[filter_index];
        output.y += pixel.x*filter_Lyy[filter_index];
        output.z += pixel.x*filter_Lxy[filter_index];
        filter_index++;
```

```
    }
    array_value_Ixx [x + y* width] = output.x;
    array_value_Iyy [x + y* width] = output.y;
    array_value_Ixy [x + y* width] = output.z;
    }
}

__kernel void convolution_vertical (__global float* array_value_Ixx_hor,
                                    __global float* array_value_Iyy_hor,
                                    __global float* array_value_Ixy_hor,
                                    __constant int* dim,
                                    __constant float* filter_Lxx,
                                    __constant float* filter_Lyy,
                                    __constant float* filter_Lxy,
                                    __global float* array_value_Ixx_ver,
                                    __global float* array_value_Iyy_ver,
                                    __global float* array_value_Ixy_ver)
        {
    int x = (int)get_global_id(0);
    int y = (int)get_global_id(1);
    int width = dim[0];
    int height = dim[1];
    float value_min = -20.06;
    float value_max = 22.14;
    float4 output = (float4)(0.0f, 0.0f, 0.0f, 0.0f);
    array_value_Ixx_ver [x + y* width] = 0.0f;
    array_value_Iyy_ver [x + y* width] = 0.0f;
    array_value_Ixy_ver [x + y* width] = 0.0f;

    if(y>=FILTER_HALF_WIDTH && y<(height-FILTER_HALF_WIDTH)){
    int filter_index = 0;
    for (int i=-FILTER_HALF_WIDTH; i<=FILTER_HALF_WIDTH; i++) {
         float pixel_Ixx = array_value_Ixx_hor[x + y* width + i* width];
         float pixel_Iyy = array_value_Iyy_hor[x + y* width + i* width];
         float pixel_Ixy = array_value_Ixy_hor[x + y* width + i* width];
         output.x += pixel_Ixx*filter_Lxx[filter_index];
         output.y += pixel_Iyy*filter_Lyy[filter_index];
         output.z += pixel_Ixy*filter_Lxy[filter_index];
         filter_index++;
         }
    output.x = 1.0/(value_max - value_min) * (output.x - value_min);
    output.y = 1.0/(value_max - value_min) * (output.y - value_min);
    output.z = 1.0/(value_max - value_min) * (output.z - value_min);
    array_value_Ixx_ver [x + y* width] = output.x;
    array_value_Iyy_ver [x + y* width] = output.y;
    array_value_Ixy_ver [x + y* width] = output.z;
    }
}

__kernel void determinant       (__global float* Ixx,
                                 __global float* Iyy,
                                 __global float* Ixy,
```

```
                                    __constant int* dim,
                                    __constant float* boundary_I,
                                    __global float* D)
{
   int x = (int)get_global_id(0);
   int y = (int)get_global_id(1);
   int width = dim[0];
   float value_min = boundary_I[0];
   float value_max = boundary_I[1];

   Ixx [x + y* width] = 1.0/(value_max-value_min)*(Ixx [x + y* width]-
      value_min);
   Iyy [x + y* width] = 1.0/(value_max-value_min)*(Iyy [x + y* width]-
      value_min);
   Ixy [x + y* width] = 1.0/(value_max-value_min)*(Ixy [x + y* width]-
      value_min);

   D [x + y* width] =   Ixx [x + y* width] * Iyy [x + y* width] - (Ixy [x
      + y* width] * Ixy [x + y* width]);
}

__kernel void neighbour_suppression     (__global float* D,
                                         __constant int* dim,
                                         __global float* feature,
                                         __global float* neighbour_supp,
                                         write_only image2d_t image_max)
{
   int x = (int)get_global_id(0);
   int y = (int)get_global_id(1);
   int width = dim[0];

   neighbour_supp [x + y* width] = 0.0f;

   float value_D = D [x + y* width];
   feature [x + y* width] = fabs(value_D);

   float temp = -1000.0f;
   if (temp < D TOP_LEFT)
       temp = D TOP_LEFT;
   if (temp < D TOP_CENTER)
       temp = D TOP_CENTER;
   if (temp < D TOP_RIGHT)
       temp = D TOP_RIGHT;
   if (temp < D LEFT)
       temp = D LEFT;
   if (temp < D RIGHT)
       temp = D RIGHT;
   if (temp < D LOWER_LEFT)
       temp = D LOWER_LEFT;
   if (temp < D LOWER_CENTER)
       temp = D LOWER_CENTER;
   if (temp < D LOWER_RIGHT)
```

```
        temp = D LOWER_RIGHT;

    if (temp < feature [x + y* width]) && temp > threshold){
         neighbour_supp [x + y* width] = feature [x + y* width];
         write_imagef(image_max, (int2)(x, y), (float4)(value_D, 1.0f, 1.0f
             , 1.0f));
         }
    else
    write_imagef(image_max, (int2)(x, y), (float4)(0.0f, 1.0f, 1.0f, 1.0f))
        ;
}
```