UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Fraunhofer FIT
Fraunhofer-Institut für Angewandte Informationstechnik FIT

Implementierung des SURF-Feature-Detektors auf der GPU mit Hilfe von CUDA

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von

Robert Hofmann

Betreuer: Dipl.-Inform. Jan Robert Menzel
Arbeitsgruppe Kollaborative Virtuelle und Augmentierte Umgebungen

Koblenz, im Oktober 2009

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....
(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

1 Einleitung

Seit der Einführung programmierbarer Prozessoren auf Grafikkarten, den sogenannten *Shadern*, sind diese zunehmend nicht nur für deren eigentlichen Zweck, dem *Graphics Processing* (GP) in der Computergrafik, eingesetzt worden. Shader ermöglichen den hohen Grad an Parallelität der Grafikprozessoren, kurz *GPU* (*graphics processing unit*), für Algorithmen auszunutzen, die eigentlich keinen Bezug zur Computergrafik haben. Viele Algorithmen, die traditionell für CPUs entwickelt wurden, sind daher inzwischen erfolgreich auf GPUs portiert worden und konnten so zum Teil erheblich beschleunigt werden. Dadurch entstand eine neues Teilgebiet in der Informatik, das als *GPGPU* (*general purpose computation on graphics processing unit*) bezeichnet wird.

Durch das Nutzen der GPU konnten unter anderem Methoden aus dem Bereich maschinelles Sehen, wie zum Beispiel die Detektion und Extraktion von markanten, lokalen Merkmalen aus einem Bild, soweit beschleunigt werden, dass sie in Echtzeit angewandt werden können.

Der wohl verbreitetste Algorithmus im Bereich Merkmalsdetektion (*feature detection*) ist *SIFT* (*scale-invariant feature transform*) von Lowe [9]. Das Vergleichen der durch *SIFT* extrahierten lokalen Bildmerkmale zwischen Bildern (*matching*) findet eine breite Anwendung in verschiedenen Aufgabenstellungen des machinellen Sehens. So zum Beispiel in der Objekterkennung, autonomen Navigation und der Bildregistrierung. Der Nachteil des *SIFT*-Algorithmus ist sein hoher Rechenaufwand. Der daraufhin von Bay et al. entwickelte *SURF*-Algorithmus (*speeded-up robust features*) [2] beschleunigte die Merkmalsextraktion durch verschiedene Approximationen zwar erheblich, war jedoch ebenso noch nicht schnell genug, um effizient in Echtzeitanwendungen genutzt werden zu können. Erst die Portierung dieser Algorithmen auf die GPU [7, 19, 5, 17] ermöglichten Frameraten von bis zu 30 Hertz für *SIFT* und 70 Hertz für *SURF*.

Der Nachteil von GPGPU-Berechnungen mit Shader-Programmen ist jedoch, dass diese nur unter Ausnutzung einer Grafikschnittstelle (wie zum Beispiel *OpenGL*) programmiert werden können. Infolge dessen muss für diese allgemeineren Berechnungsprobleme eine Grafikanwendung gewissermaßen zweckentfremdet und das Problem dementsprechend anders modelliert werden. Um das Programmieren von GPUs mehr zu generalisieren und die verschiedenen GPU-Typen in einem gemeinsamen Programmiermodell zu abstrahieren, wurden Architekturen entwickelt, die eine Benutzung der GPU als eine Art Co-Prozessor erlauben und mit erweiterten Varianten der Hochsprache *C* programmiert werden können [16, 12, 1]. Eine dieser Architekturen ist *CUDA* (*compute unified device architecture*). Mit Hilfe von *CUDA* können die GPUs von *Nvidia*-Grafikkarten programmiert werden ohne auf Shader-Programme zurückgreifen zu müssen.

Das Ziel dieser Arbeit war es den *SURF*-Detektor vollständig mittels *CUDA* zu implementieren, um ihn so gegenüber einer CPU-Implementation zu beschleunigen. Die Details der Implementation sowie die Ergebnisse werden in dieser Arbeit dokumentiert. Dazu wird zu Beginn in Abschnitt 2 der Detektor des *SURF*-Algorithmus ausführlich beschrieben. Danach folgt in Abschnitt 3 ein Überblick über den Aufbau und das Programmiermodell von *CUDA*. Abschnitt 4 umfasst eine detaillierte Erklärung der Implementation sowie eine Auswertung der Ergebnisse.

1.1 Verwandte Arbeiten

Es existieren zahlreiche Implementationen des *SURF*-Algorithmus für die CPU. Neben der nicht quelloffenen Bibliothek der *SURF*-Entwickler, wurde *SURF* zum Beispiel von Liu et al. auch in *OpenCV* [3] integriert. Weiterhin gibt es noch die gut dokumentierte und quelloffene Implementation *OpenSURF* von Evans [6].

Um die Korrektheit der Implementation dieser Arbeit überprüfen zu können, war es wichtig ihre Ausgaben, sowie die Ausgaben der Zwischenschritte mit einer CPU-Referenzimplementierung vergleichen zu können. Sollten sich Unterschiede zu dieser Referenz ergeben, die nicht auf Programmierfehler zurückzuführen sind, so ist genau zu klären welche Ursache diese haben. Aufgrund der Verfügbarkeit des Quellcodes und der ausführlicheren Dokumentation wurde als Referenz *OpenSURF* ausgewählt.

Von den *SURF*-Entwicklern stammt ebenso eine nicht quelloffene GPU-Portierung [5] ihres Algorithmus. Mit dieser konnte *SURF* erheblich beschleunigt werden. Da der Algorithmus der GPU-Implementation sich teilweise erheblich von der ursprünglichen Beschreibung in [2] und damit auch von deren CPU-Version unterscheidet, sind die Ausgaben beider Algorithmen zwar vergleichbar, aber jedoch nicht mehr identisch. Teilweise wurde diese *SURF*-Variante unter Verwendung von *CUDA* implementiert. Für den Großteil des Detektor-Schrittes wurden jedoch Shader-Programme verwendet. Insbesondere wurde dort auf computergrafik-spezifische Instrumente, wie zum Beispiel Mip-Mapping, zurückgegriffen.

Von Terriberry et al. [17] stammt eine weitere Portierung des *SURF*-Algorithmus auf die GPU. Diese wurde vollständig mit Shader-Programmen realisiert. Viele der nötigen Berechnungen wurden dabei mit komplexen Multipass-Operationen optimiert. In einigen Punkten weicht auch hier die Implementation vom originalem *SURF*-Algorithmus ab. Ein Vergleich mit den Ausgaben der originalen *SURF*-Bibliothek fand jedoch nicht statt.

2 SURF-Detektor

Der *SURF*-Algorithmus von Bay et al. [2] ist ein Algorithmus zum effizienten Detektieren von Interessenpunkten (*interest points*) in einem Bild und zur Berechnung eines Deskriptors zu jedem dieser Interessenpunkte. Die Position des Interessenpunktes und dessen *SURF*-Deskriptor bilden ein *SURF*-Merkmal (*SURF feature*). Der Deskriptor ist dabei für jedes *SURF*-Merkmal einzigartig, aber dennoch auch invariant gegenüber Skalierung, Rotation, Beleuchtungsänderung, Bildrauschen und bis zu einem gewissen Grad auch gegen perspektivische Verzerrung.

Die Funktionsweise von *SURF* ist angelehnt an den *SIFT*-Algorithmus, ist in der Berechnung jedoch deutlich schneller. Ein Großteil des Geschwindigkeitsvorteils von *SURF* ist auf die Verwendung von Integralbildern (*integral image*, *summed area table*) [18] zurückzuführen.

Im Folgenden werden die einzelnen Elemente des *SURF*-Detektors näher beschrieben. Die Berechnung des *SURF*-Deskriptors und das Vergleichen von *SURF*-Merkmalen ist nicht Gegenstand dieser Arbeit und wird hier daher nicht näher betrachtet.

2.1 Integralbilder

In einem Vorverarbeitungsschritt wird aus dem Eingabebild ein Integralbild [18] berechnet. In einem Integralbild ist jeder Pixelwert die Summe aller Grauwerte zwischen dem korrespondierendem Pixel im Eingabebild und dem Ursprung. Seien ein Bild I und die Pixelkoordinate (x, y) gegeben. Der Wert im Integralbild I_Σ an der Position (x, y) ist dann formal definiert durch:

$$I_\Sigma(x, y) = \sum_{i=0}^{i \leq x} \sum_{j=0}^{j \leq y} I(i, j) \ . \tag{1}$$

Die Summe der Pixelwerte eines beliebigen achsenparallelen Rechtecks in Bild I lässt sich dann mit lediglich drei Additionen und vier Speicherzugriffen auf I_Σ berechnen. Dieses Vorgehen ist schematisch in Abbildung 1 dargestellt.

Formal ist der Zusammenhang zwischen einer Rechtecksumme in einem Bild und dessen Integralbild wie folgt definiert (Variablennamen wie in Abbildung 1):

$$\begin{aligned} \Sigma &= \sum_{i=A_x}^{i \leq D_x} \sum_{j=A_y}^{j \leq D_y} I(i, j) \\ &= I_\Sigma(D) - I_\Sigma(B) - I_\Sigma(C) + I_\Sigma(A) \ . \end{aligned} \tag{2}$$

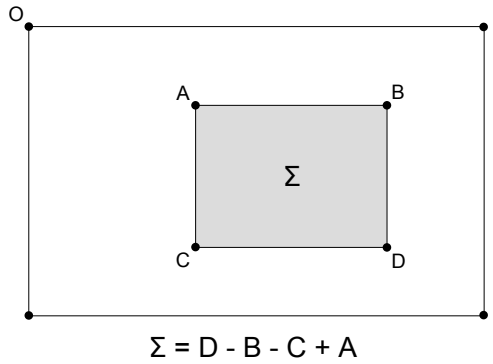

Abbildung 1: Berechnung der Summe aller Pixelwerte im grau markiertem Rechteck mit Hilfe eines Integralbildes. Vgl. [2]

2.2 Beschleunigter Hesse-Detektor

Zur Detektion von Interessenpunkten wird im Bild nach Positionen gesucht an denen die Determinante der Hesse-Matrix ein lokales Maximum erreicht. Um die Elemente der Hesse-Matrix an einer Position (x, y) im Eingabebild zu berechnen, wird es bei *SURF* mit Marr-Hildreth-Operatoren, besser bekannt als *Laplacian of Gaussians* (*LoG*), für die x-, y- und xy-Richtung gefalten. Diese Operatoren werden durch Anwenden des Laplace-Operators auf eine Gauß-Funktion berechnet, sind also im Grunde partielle Ableitungen zweiter Ordnung von dieser. Durch die Verwendung der Gauß-Funktion kann mittels der Varianz σ die Glättung des Eingabebildes beeinflusst und somit die Determinante auf verschiedenen Skalierungsebenen des Bildes berechnet werden. Auf diese Weise wird die Skalierungsinvarianz eines *SURF*-Features gewährleistet. Nähere Erläuterungen hierzu folgen in Abschnitt 2.3.

Die Hesse-Matrix lässt sich nun als eine Funktion $\mathcal{H}$ der Position $\mathbf{x} = (x, y)$ und der Skalierung (*scale*) σ definieren:

$$\mathcal{H}(\mathbf{x}, \sigma) = \begin{bmatrix} L_{xx}(\mathbf{x}, \sigma) & L_{xy}(\mathbf{x}, \sigma) \\ L_{xy}(\mathbf{x}, \sigma) & L_{yy}(\mathbf{x}, \sigma) \end{bmatrix} , \tag{3}$$

wobei $L_{xx}(\mathbf{x}, \sigma)$ die Faltung der zweiten partiellen Ableitung der Gauß-Funktion $\frac{\partial^2}{\partial x} g(\sigma)$ nach x mit dem Eingabebild an Position $\mathbf{x}$ ist. L_{yy} und L_{xy} sind analog dazu definiert.

Ob sich im Bild an der Position $\mathbf{x}$ ein Extremum, also ein möglicher Interessenpunkt befindet, lässt sich an der Determinante der Hesse-Matrix ablesen:

$$\det(\mathcal{H}) = L_{xx} L_{yy} - L_{xy}^2 \ . \tag{4}$$

Ist diese negativ, so handelt es sich um einen Sattelpunkt. Ist sie positiv, so wurde ein Extremum gefunden. Nähere Informationen über das Extre-

mum liefert die Spur der Hesse-Matrix, welche mit dem Laplace-Operator korrespondiert:

$$\mathrm{tr}(\mathcal{H}) = L_{xx} + L_{yy} \ . \tag{5}$$

Das Vorzeichen des Werts der Spur gibt Auskunft über den Kontrast des Extremums, also ob es sich um einen hellen Blob auf dunklem Hintergrund oder einen dunklen Blob auf hellem Hintergrund handelt.

Für die Faltung mit einem Bild werden L_{xx}, L_{yy} und L_{xy} in der Regel in Filtermasken diskretisiert. Bay et al. schlagen für *SURF* eine noch weitergehende Approximation durch die Mittelwertfilter (*box filter*) D_{xx}, D_{yy} und D_{xy} vor, da diese schneller berechnet werden können. In Abbildung 2 sind die diskretisierten LoG-Filter und ihre entsprechenden Approximationen dargestellt. Um die Summen der Filter unabhängig von der Filtergröße zu berechnen, werden sie bezüglich der Filterfläche normiert.

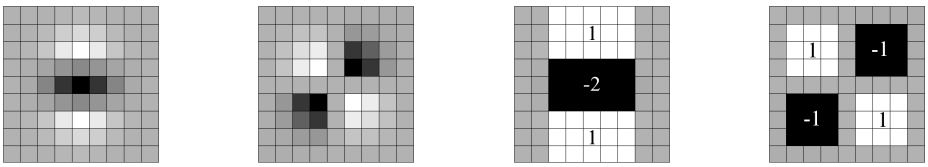

Abbildung 2: Von links nach rechts: Die diskretisierten Filtermasken für die zweiten partiellen Ableitung der Gauß-Funktion in y- (L_{yy}) und xy-Richtung (L_{xy}) und deren in *SURF* angewandten Approximationen D_{yy} und D_{xy}. Die grauen Regionen entsprechen einer Gewichtung mit 0. [2]

Die Determinante der so approximierten Hesse-Matrix lässt sich durch folgende Formel berechnen:

$$\det(\mathcal{H}_{approx}) = D_{xx}D_{yy} - (wD_{xy})^2 \ , \tag{6}$$

wobei w ein Gewicht ist, um den Unterschied der approximierten Filtermasken zu den ursprünglichen auszugleichen. Das Gewicht w variiert geringfügig abhängig von der Filtergröße, ist in *SURF* aber ein konstanter Wert ($w = 0,9$). Der Wert der Determinante wird als *Blob* an Position $\mathbf{x} = (x, y, \sigma)^T$ bezeichnet.

2.3 Konstruktion des Skalenraumes

Interessenpunkte müssen, damit Skalierungsinvarianz gewährleistet werden kann, in verschiedenen Skalierungsebenen detektiert werden. In *SIFT* werden diese Skalierungsebenen in einer Skalenraum-Pyramide (*scale-space pyramid*) implementiert: Durch iteratives Gaußfiltern und Verkleinern des Eingabebildes wird eine Pyramide von verschiedenen Skalierungsebenen

des Bildes erstellt. In *SURF* hingegen wird nicht das Bild verkleinert, sondern werden die Filtermasken zur Berechnung der Blob-Werte für jede Skalierungsebene entsprechend vergrößert. Da die approximierten LoG-Filter mit Hilfe des Integralbildes unabhängig von ihrer Größe in nahezu konstanter Zeit berechnet werden können, führt diese Methodik zu keinen erhöhten Rechenaufwand. Abbildung 3 veranschaulicht diese unterschiedlichen Vorgehensweisen.

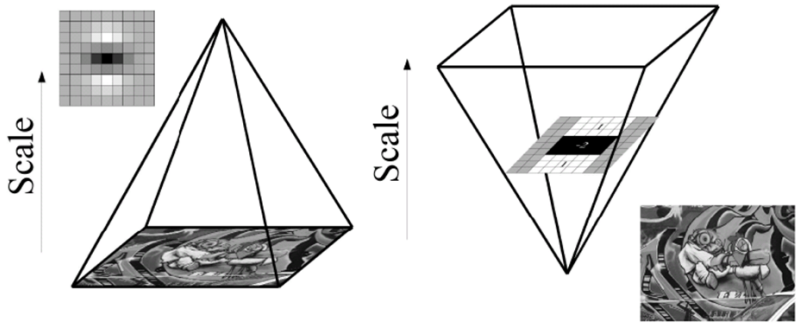

Abbildung 3: Anstatt das Eingabebild iterativ zu verkleinern (links), erlauben Integralbilder das Anwenden eines iterativ wachsenden Filters (rechts) auf das gleich bleibende Bild bei konstantem Aufwand. [2]

Der Skalenraum ist unterteilt in *Oktaven*. Eine Oktave repräsentiert eine Reihe von Filterungen des Eingabebildes mit wachsender Filtergröße und umfasst einen Skalierungsfaktor von mindestens 2. Jede Oktave ist in eine konstante Anzahl von Skalierungsebenen, sogenannten *Intervallen*, unterteilt, die jeweils eine schrittweise Erhöhung der Skalierung repräsentieren.

Die 9×9 Filtermasken D_{xx}, D_{yy} und D_{xy} (Abbildung 2) approximieren die LoGs mit der Varianz $\sigma = 1,2$ und berechnen die Blob-Werte für das erste Intervall der ersten Oktave mit einer Skalierung von $1,2$. Vergrößern dieser LoG-Filter vergrößert auch die Skalierung. Vorausgesetzt, dass das Filter-Layout im Verhältnis konstant bleibt, lässt sich die Skalierung aus der Filtergröße berechnen:

$$\begin{aligned} \sigma &= \text{aktuelle Filtergröße} \cdot \frac{\text{Bezugs-Filterskalierung}}{\text{Bezugs-Filtergröße}} \\ &= \text{aktuelle Filtergröße} \cdot \frac{1,2}{9} \quad . \end{aligned} \tag{7}$$

Da die LoG-Filter diskretisiert sind und ein Zentrumspixel (d.h. eine ungerade Maskengröße) haben müssen, lassen sie sich nur in eingeschränkten Schrittweiten vergrößern. Diese Schrittweite, und damit auch die möglichen Skalierungsschritte, sind abhängig von der Größe der Teilflächen[1]

[1] Diese Teilflächen werden in [2] als *lobes* (eng., „Flügel“, „Lappen“) bezeichnet.

der LoG-Filter (in Abbildung 2 schwarz oder weiß gekennzeichnet). Die jeweils kürzere Länge einer Teilfläche eines Filters beträgt stets ein Drittel der Filtergröße und wird als l bezeichnet. Verdeutlicht ist dies in Abbildung 4: Für die initialen LoG-Filter mit der Filtergröße $w_0 = 9$ (linke Seite der Abbildung), ist die Länge der Teilflächen $l_0 = 3$. Um das Zentrumspixel der schwarzen Teilfläche in D_{yy} (oben links) zu erhalten, muss l_0 um mindestens 2 Pixel vergrößert werden. Da sich in D_{yy} insgesamt drei dieser Teilflächen befinden, ergibt sich für w_0 eine Vergrößerung von 6 Pixeln auf $w_1 = 15$. Daraus resultiert nach Gleichung 7 für das zweite Intervall der ersten Oktave eine Skalierung von $\sigma_1 = 2 = 15\frac{1,2}{9}$. Die Filtergrößen der nächsten Intervalle sind $w_2 = 21$ mit der Skalierung $\sigma_2 = 2,8$ und $w_3 = 27$ mit $\sigma_3 = 3,6$. Insgesamt sind das vier Intervalle pro Oktave, die in der ersten Oktave eine Skalierung von $1,2$ bis $3,6$ umfassen.

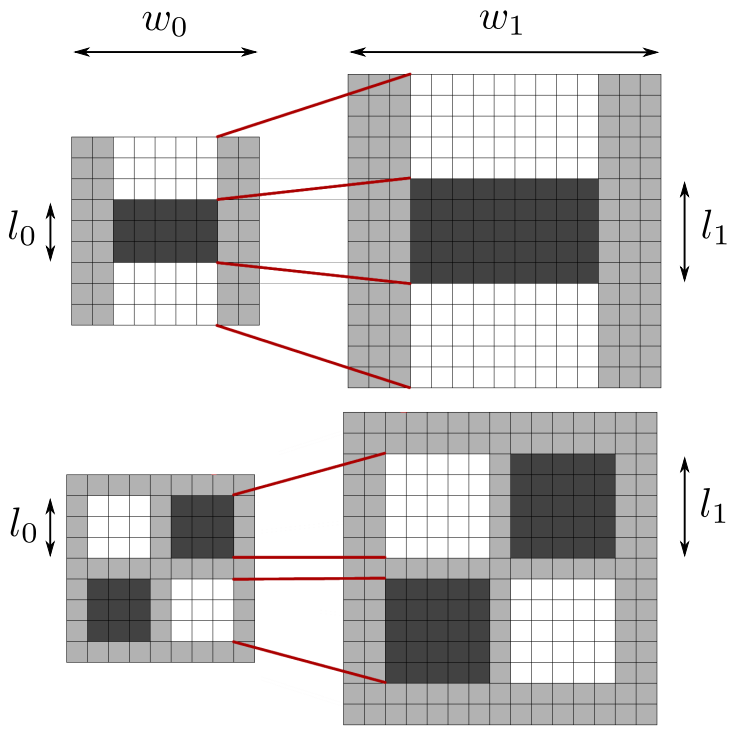

Abbildung 4: Um das Filter-Layout zu erhalten, dürfen die LoG-Filter jeweils nur um 6 Pixel oder mehr vergrößert werden. Vgl. [2]

Für die folgenden Oktaven wird der Vergrößerungsschritt der Filtergröße w jeweils verdoppelt. In der zweiten Oktave von 6 Pixel auf 12, in der dritten von 12 auf 24 und so weiter. Daraus ergeben sich die in Abbildung 5 dargestellten Filtergrößen.

Zusätzlich wird mit jeder neuen Oktave die Samplingrate auf das Eingabebild halbiert bzw. der *Samplingintervall* verdoppelt.

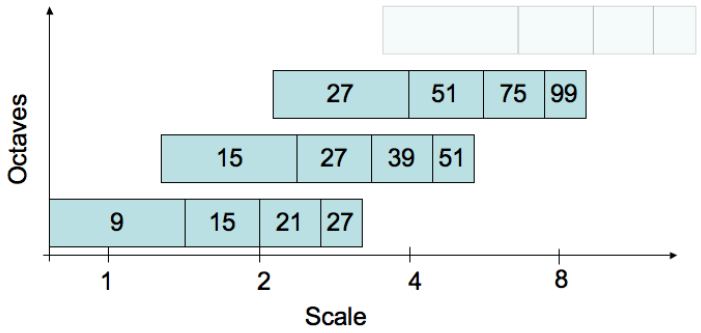

Abbildung 5: Die verschiedenen Filtergrößen in den einzelnen Oktaven. [2]

2.4 Suche lokaler Extrema

Es gilt nun innerhalb der Oktaven der Skalenraum-Pyramide Interessenpunkte zu finden. Als „interessant“ gelten lokale Extrema im Skalenraum. Dazu wird die $3 \times 3 \times 3$ Nachbarschaft jedes Blobs untersucht: Ist sein Blob-Wert betragsmäßig größer als alle seinen 26 Nachbar-Blobs, so ist er ein lokales Extremum. Ist sein Blob-Wert ebenfalls größer als ein vordefinierter Schwellwert, dann wird diese Skalenraum-Position $\mathbf{x} = (x, y, \sigma)^T$ als ein Interessenpunkt vermerkt. Abbildung 6 veranschaulicht wie sich eine $3 \times 3 \times 3$ Nachbarschaft im Skalenraum zusammensetzt.

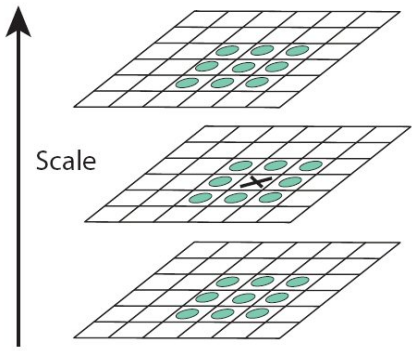

Abbildung 6: Die $3 \times 3 \times 3$ Nachbarschaft eines Blobs im Skalenraum, bestehend aus den acht räumlichen Nachbarn und den jeweils neun Nachbarn der nächstgrößeren und der nächstkleineren Skalierungsebene. [6]

2.5 Präzise Ortsbestimmung der Interessenpunkte

Im finalen Schritt des *SURF*-Detektors werden die Positionen der gefundenen Interessenpunkte auf Subpixel-Genauigkeit interpoliert, was vor allem in höheren Skalierungsebenen nötig ist, um die Interessenpunkte präzise zu lokalisieren.

Das dazu angewandte Verfahren [4] besteht darin die Hesse-Matrix als ein Funktion $\mathcal{H}(\mathbf{x})$ über Skalenraum-Koordinaten $\mathbf{x} = (x, y, \sigma)^T$ durch eine Taylor-Reihe mit den Koordinaten $\mathbf{x}_e$ des detektieren Interessenpunktes als Entwicklungspunkt anzunähern:

$$\mathcal{H}(\mathbf{x}) = \mathcal{H}(\mathbf{x}_e) + \left(\frac{\partial \mathcal{H}(\mathbf{x}_e)}{\partial \mathbf{x}}\right)^T \Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^T \left(\frac{\partial^2 \mathcal{H}(\mathbf{x}_e)}{\partial \mathbf{x}^2}\right) \Delta\mathbf{x} \ , \qquad (8)$$

wobei $\Delta\mathbf{x}$ der Abstand (*offset*) zwischen der ursprünglichen (diskreten) und der interpolierten Position des Interessenpunktes ist. Dieser Abstand lässt sich durch Ableiten dieser Funktion und Gleichsetzen mit 0 berechnen. Nach Umformung ergibt sich das lineare System

$$\Delta\mathbf{x} = -\left(\frac{\partial^2 \mathcal{H}(\mathbf{x}_e)}{\partial \mathbf{x}^2}\right)^{-1} \frac{\partial \mathcal{H}(\mathbf{x}_e)}{\partial \mathbf{x}} \ , \qquad (9)$$

aus dessen Lösungsvektor $\Delta\mathbf{x}$ sich mit

$$\hat{\mathbf{x}} = \mathbf{x}_e + \Delta\mathbf{x} \qquad (10)$$

die interpolierte Position $\hat{\mathbf{x}}$ des Interessenpunktes bei $\mathbf{x}_e$ berechnen lässt. Die Ableitungen in den obigen Gleichungen werden als Differenzen zu den umgebenden Pixeln von $(\mathbf{x}_e)$ approximiert.

Ist der Abstand $\Delta\mathbf{x}$ eines Interessenpunktes zu seiner interpolierten Position in einer der Dimensionen x, y oder σ nicht kleiner als $0,5$, so wird er als zu kontrastarm (d.h. als zu empfindlich gegenüber Rauschen) verworfen.

Die finale Ausgabe des *SURF*-Detektors ist eine Liste von Interessenpunkten mit subpixel-genauen Positionen im Eingabebild und einer zugeordneten Skalierung.

3 CUDA

CUDA ist eine von *Nvidia* entwickelte Parallel-Computing-Architektur, mit der GPUs von *Nvidia*-Grafikkarten (ab GPUs vom Typ *G8x*) programmiert werden können. Ziel ist es, Programmierern die Möglichkeit zu geben das Parallelisierungspotential von GPUs auszunutzen, ohne sich zu sehr mit

den Besonderheiten und Beschränkungen einer GPU-Architektur befassen zu müssen.

Als Programmiersprache für *CUDA*-Programme wurde *C for CUDA* entwickelt, welche eine Erweiterung der Sprache *C* ist. Durch eine geringe Anzahl von zusätzlichen Schlüsselwörtern können so innerhalb regulärem *C*-Programmcodes Teilroutinen als GPU-Code deklariert und aufgerufen werden. Als Compiler wird der von *Nvidia* für *CUDA* entwickelte *NVCC* (*Nvidia CUDA compiler*) verwendet.

3.1 Programmiermodell

Dem *CUDA*-Programmiermodell [13] zufolge wird der GPU-Code eines *CUDA*-Programms auf einem physisch separatem *Device* (engl., „Gerät“) ausgeführt, während der Rest des Programms auf dem *Host* (engl., „Gastgeber“) läuft. Das Device fungiert als ein Co-Prozessor des Hosts und verfügt über einen eigenen Speicher. Der Device-Speicher und die Datentransfers zwischen diesem und dem Host-Speicher werden vom Host mit Hilfe der *CUDA*-API (*CUDA application programming interface*) verwaltet. Wird GPU-Code auf dem Device ausgeführt, so wird dieser von einer in der Regel sehr großen Anzahl von *CUDA-Threads* parallel bearbeitet.

3.1.1 CUDA-Kernel

Programmteile eines *CUDA*-Programms, die für die Ausführung auf dem Device bestimmt sind, werden in sogenannten *Kerneln* gekapselt. Ein Beispiel für einen Kernel ist in Quellcode 1 gegeben. Dieser einfach aufgebaute Kernel liest jedes Element eines Eingabe-Arrays, quadriert es und schreibt das Ergebnis in ein Ausgabe-Array. Das Schlüsselwort `__global__` ist ein Funktionstyp-Qualifizierer (*function type qualifier*) der Funktion `squareArray()` und deklariert diese als einen *CUDA*-Kernel. Daraus ergeben sich mehrere Implikationen.

- Da Kernel auf dem Device ausgeführt werden, haben sie auch nur Zugriff auf den Speicher des Devices. Sämtliche Eingabedaten müssen folglich zuvor explizit vom Host-Speicher in den Device-Speicher kopiert werden. Die Funktionsparameter des Kernels werden jedoch beim Aufruf implizit kopiert.
- Ebenso müssen auch die Ausgabedaten eines Kernels explizit vom Device- in den Host-Speicher kopiert werden.
- Kernel haben immer den Rückgabetyp `void`.
- Rekursion und das Aufrufen anderer Kernel ist in einem Kernel nicht gestattet.

```
1 __global__
2 void squareArray(float* in, float* out, int n) {
3   int idx = blockIdx.x * blockDim.x + threadIdx.x;
4
5   if (idx < n) {
6     float e = in[idx];
7     out[idx] = e * e;
8   }
9 }
```

Quellcode 1: Ein einfacher *CUDA*-Kernel, der jedes Element eines Arrays quadriert.

Ein Kernel definiert den Ausführungspfad eines *CUDA*-Threads, wird aber in der Regel von vielen Threads gleichzeitig und parallel ausgeführt. Anhand des Index eines Threads, dem *Thread-Index*, können der Zugriff auf Ein- bzw. Ausgabedaten geregelt und Programmentscheidungen getroffen werden. So wird beispielsweise in Quellcode 1, Zeile 3 für jeden Thread die fortlaufende Nummer `idx` berechnet. Threads mit aufeinander folgenden Indizes greifen dementsprechend auf hintereinander liegende Elemente in den Arrays `in` und `out` zu. Da die gesamte Anzahl an Threads, die einen Kernel ausführen, in Thread-Gruppen, sogenannten *Blöcken* (*blocks*), aufgeteilt sind, wird zur Berechnung von `idx` der Index des Threads relativ zum Block, die Größe des Blocks sowie der jeweilige Index des Blocks verwendet. Eine detaillierte Beschreibung dieser Hierarchie wird im Abschnitt 3.1.3 gegeben.

3.1.2 Device-Funktionen

Eine weiterer Bestandteil von *C for CUDA* sind *Device-Funktionen*. Diese werden mit dem Funktionstyp-Qualifizierer `__device__` deklariert und sind die einzigen Funktionstypen die innerhalb von Kerneln aufgerufen werden können. Anders als Kernel dürfen Device-Funktionen auch Werte zurückgeben, können jedoch nicht vom Host aus aufgerufen werden.

Wird eine Funktion sowohl als `__device__` als auch mit dem Schlüsselwort `__host__` deklariert, so wird diese Funktion jeweils einmal für die GPU und ein weiteres mal für die CPU kompiliert, um sie so auf dem Device und dem Host ausführbar zu machen.

Standardmäßig werden Device-Funktionen als Inline-Funktionen behandelt. Mit dem Funktionstyp-Qualifizierer `__noinline__` wird dieses Verhalten unterbunden.

3.1.3 Ausführungsmodell

Die Anzahl der Threads die einen Kernel bearbeiten wird der GPU zur Laufzeit beim Aufruf des Kernels mitgeteilt. Um die Berechnungen effi-

zient auf die Multiprozessoren der GPU aufteilen zu können, werden alle Threads in *Thread-Blöcken* (*thread blocks*) organisiert. Die Gesamtheit dieser Blöcke wird als *Grid* (engl., „Netz“, „Raster“) bezeichnet. Abbildung 7 veranschaulicht diese Hierarchie. Mehrere Threads werden in einen maximal dreidimensionalem Block zusammengefasst. Relativ zu seinen Block verfügt jeder Thread über einen *Thread-Index*, der innerhalb eines Kernels über die intrinsische (*built-in*) Variable `threadIdx` erfragt werden kann. Die Ausdehnung eins Blocks ist durch die intrinsischen Variable `blockDim` für jeden Thread verfügbar. Im Kernel-Grid werden alle Thread-Blöcke als ein ein- oder zweidimensionales Raster organisiert, in dem jeder Block über einen *Block-Index* verfügt. In den Variablen `blockIdx` und `gridDim` sind der Index des aktuellen Blocks und die Ausdehnung des Grids gespeichert.

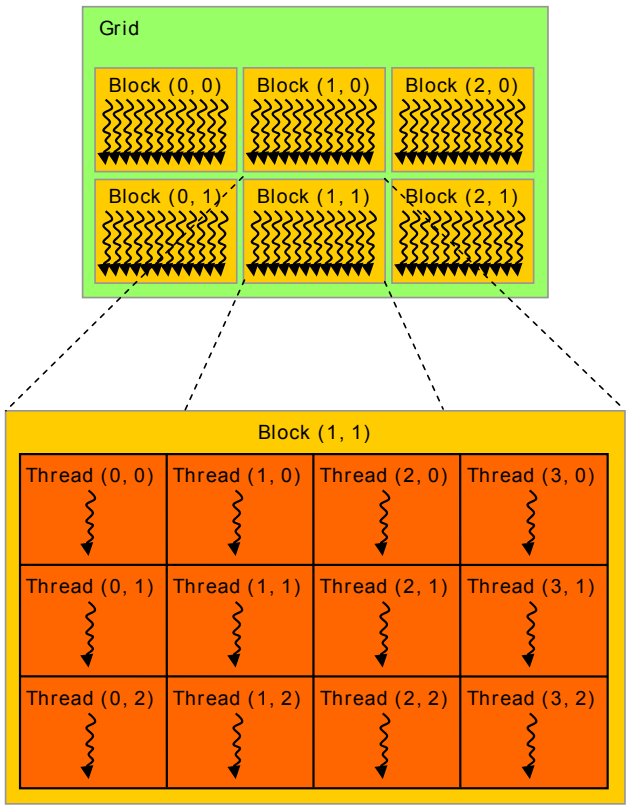

Abbildung 7: Ein Kernel-Grid ist aufgeteilt in Blöcke, welche Gruppen von Threads darstellen. Jeder Thread hat relativ zu seinem Block einen maximal dreidimensionalen Index, jeder Block einen maximal zweidimensionalen Index relativ zum Grid. [13]

Intern ist jeder *CUDA*-Thread mit einer fortlaufenden, eindeutigen Identifikationsnummer, der *Thread-ID*, versehen. Die Variablen `threadIdx` und `blockIdx` sind zur Vereinfachung jedoch vom Vektortyp `uint3`, also dreidimensionale, vorzeichenlose Ganzzahl-Variablen. Die Variablen `block-`

`Dim` und `gridDim` sind vom Vektortyp `dim3`, welcher ein `uint3` darstellt, dessen Werte automatisch mit 1 initialisiert werden. Der Zusammenhang zwischen der eindimensionalen ID eines Threads und seinem mehrdimensionalem Index ist durch die folgende Formel gegeben:

$$\begin{aligned} threadID = threadIdx.x + threadIdx.y \cdot blockDim.x \\ + threadIdx.z \cdot blockDim.x \cdot blockDim.y \ . \end{aligned} \tag{11}$$

Wie viele Threads einen Kernel bearbeiten und wie diese in Blöcke aufgeteilt werden, legt die *Ausführungs-Konfiguration* (*execution configuration*) eines Kernel-Aufrufs fest. Die *C*-Syntax für Funktions-Aufrufe wurde dafür in folgender Weise um ein `<<<...>>>`-Konstrukt erweitert:

```
kernelName<<<dim3 gridDim, dim3 blockDim>>>(...);
```

Zwischen Kernel-Name und dessen Parameterliste werden, eingefasst in drei Winkelklammerpaare, die Größe des Grids und der Blöcke an den Kernel mit übergeben. Ein Beispiel für den Aufruf eines Kernels ist in Quellcode 2 gegeben. Dort werden die Ausdehnung des Grids und der Blöcke in den Variablen `numBlocks` und `blockSize` gespeichert und an den Kernel `squareArray()` aus Quellcode 1 übergeben. Dazu wird die erste Dimension der Variable `blockSize` mit 512 initialisiert (ein Block ist also eine eindimensionalen Gruppe von 512 Threads). Abhängig von der Anzahl der Elemente im Eingabe-Array `d_in` (im Beispiel gespeichert in der Variable `size`) wird dann die Größe des Grids, also die Anzahl der nötigen Blöcke berechnet, um jedes Element des Arrays zu bearbeiten. Da `d_in` beliebig groß sein kann, sich also eventuell nicht gleichmäßig auf n Blöcke à 512 Threads aufteilen lässt, muss ein zusätzlicher Block erzeugt werden, falls `size` kein Vielfaches von 512 ist. Ein Teil der Threads dieses Blocks wird in diesem Fall also kein Array-Element zugeordnet werden können und daher im Kernel nicht genutzt werden.

```
1 dim3 numBlocks, blockSize(512);
2 numBlocks.x = size % blockSize == 0 ?
3               size / blockSize : size / blockSize + 1;
4 squareArray<<<numBlocks, blockSize>>>(d_in, d_out, size);
```

Quellcode 2: Aufruf des Kernels `squareArray()` aus Quellcode 1.

Wie viele Threads in einem Block zusammengefasst werden können, ist durch die Hardware beschränkt. Die maximalen Größen der einzelnen Dimensionen eines Blocks sind wie folgt: 512 in der x-, 512 in der y- und 64 in der z-Dimension. Insgesamt darf ein Block jedoch maximal nur 512 Threads umfassen. Die maximale Größe des Grids ist 65535 in der x- und 65535 in der y-Dimension.

Der Nutzen dieser Block-Hierarchie ist die Möglichkeit, die einzelnen Blöcke unabhängig voneinander rechnen lassen zu können. Je nachdem über wie viel Multiprozessoren eine GPU verfügt, werden alle Blöcke gleichmäßig auf diese verteilt. Eine GPU mit vielen Multiprozessoren wird also mehr Blöcke parallel bearbeiten können als eine GPU mit weniger Multiprozessoren, die mehr Blöcke sequentiell bearbeiten muss. Abbildung 8 veranschaulicht dies. Heutige *CUDA*-Programme können also von zukünftigen, leistungsstärkeren GPUs profitieren ohne direkt auf diese angepasst worden zu sein. Diese Eigenschaft wird auch als Skalierbarkeit (*scalability*) bezeichnet.

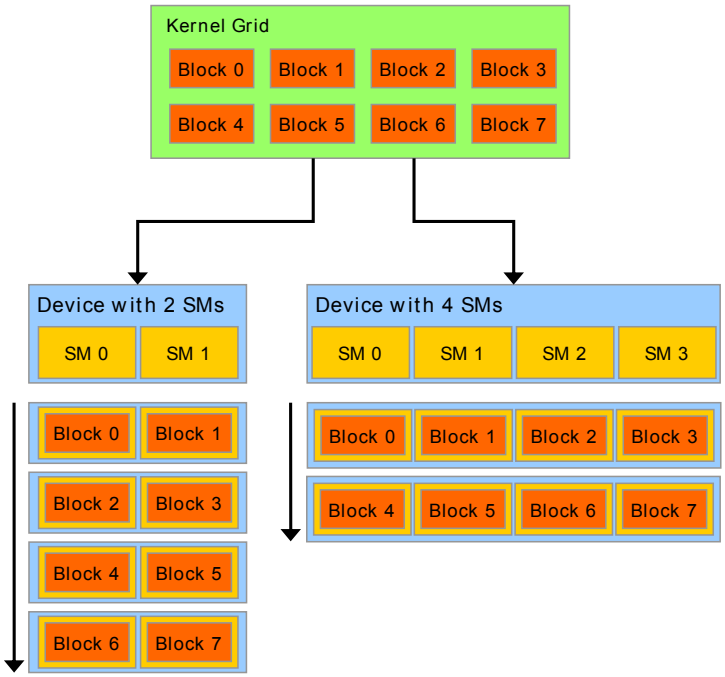

Abbildung 8: Das Kernel-Grid (oben) wird auf einer GPU mit vier Multiprozessoren (rechts) besser parallelisiert als auf einer GPU mit lediglich zwei Multiprozessoren (links). Der theoretische Beschleunigungsfaktor beträgt 2. [13]

3.1.4 Speichermodell

Analog zu der hierarchischen Strukturierung der Threads eines Kernels (Abbildung 7), ist der Zugriff auf den Device-Speicher auch in einer Hierarchie organisiert. Abbildung 9 stellt diese beiden Hierarchien gegenüber. Jeder Thread hat einen privaten *lokalen Speicher* (*local memory*) mit der Lebensdauer des Threads. Alle Threads eines Blocks haben Zugriff auf einen *Shared-Speicher* (von: „shared“, engl., „geteilt“, „verteilt“) mit der Lebensdauer des Blocks. Jeder Kernel eines *CUDA*-Programms hat Zugriff auf den

globalen Speicher (*global memory*), dessen Lebensdauer über Kernel-Aufrufe hinweg besteht und durch die Laufzeit des ganzen Programms bestimmt ist.

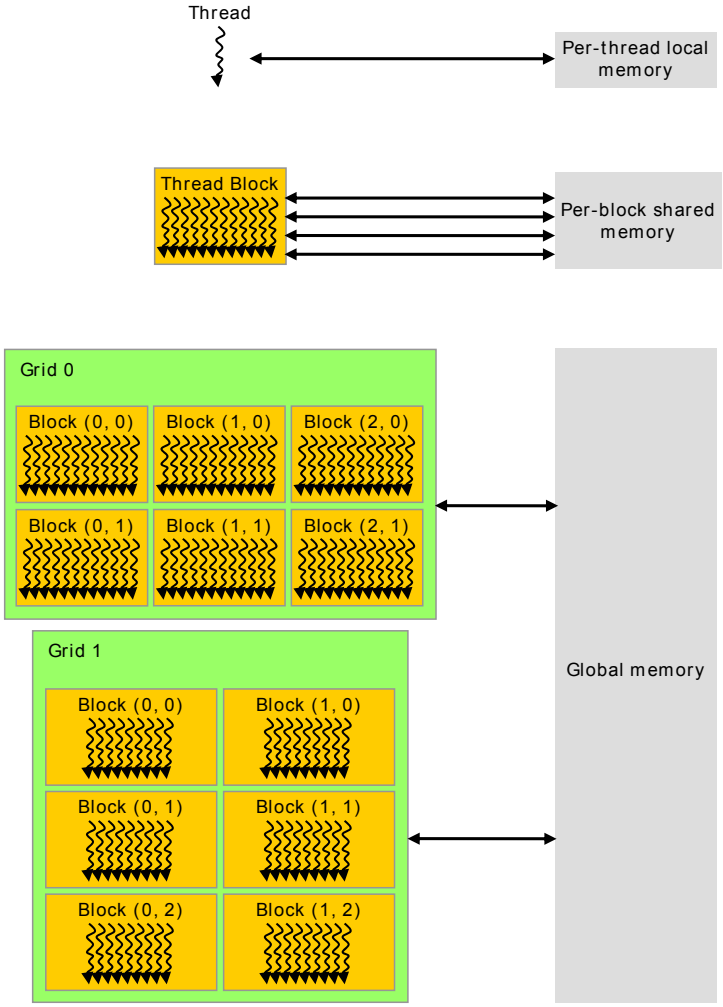

Abbildung 9: Ähnlich der Thread-Hierarchie gibt es in *CUDA* eine entsprechende Speicherhierarchie. [13]

Die verschiedenen Arten des Device-Speichers sowie deren Eigenschaften werden hier nun kurz dargelegt. Dazu ist in Abbildung 10 das Speichermodell von *CUDA* zusätzlich auch visualisiert.

Register (*registers*): Stellen einen schnellen Speicher (Latenzzeit: 0 Taktzyklen) für lokale Variablen in Kerneln bereit. Nur der ausführende Thread hat Zugriff auf seinen Register-Speicher. Die Lebensdauer von Daten im Register-Speicher entspricht der des Threads. Die Anzahl an verfügbaren Registern für jeden Thread ist abhängig von der

Anzahl an Threads pro Block, da für jeden Block nur eine begrenzte Anzahl an 32-Bit-Register zur Verfügung stehen.

Lokaler Speicher (*local memory*): Ist ein vergleichsweise langsamer Speicher (Latenzzeit: 400-600 Taktzyklen), in dem automatisch private Variablen eines Threads gespeichert werden, die zu groß für den Register-Speicher sind.

Shared-Speicher (*shared memory*): Ist ein sehr schneller Speicher (Latenzzeit: bis zu 0 Taktzyklen), der für jeden Block existiert und auf dem alle Threads dieses Blocks Zugriff haben. In ihm sollten Daten gespeichert werden, auf die mehrere Threads schnellen Zugriff benötigen. Des weiteren kann er zur Kommunikation von Zwischenergebnissen unter den Threads eines Blocks genutzt werden. Die Lebensdauer von Daten im Shared-Speicher entspricht der des Blocks dem er zugeordnet ist.

Globaler Speicher (*global memory*): Ist ein langsamer Speicher (Latenzzeit: 400-600 Taktzyklen), auf den alle Threads eines Kernels Zugriff haben. Die Lebensdauer von Daten im globalen Speicher entspricht der des ausgeführten *CUDA*-Programms. Der globale Speicher ist die Datenverbindung zwischen Host und Device: Der Host kann globalen Speicher allozieren und Datenmengen hinein- bzw. herauskopieren und das Device kann Daten ab einer gegeben Speicheradresse lesen und schreiben.

Konstanter Speicher (*constant memory*): Ist ein Nur-Lese-Speicher mit Cache. Die Zugriffszeit ist gleich der vom globalen Speicher (also 400-600 Taktzyklen) bei einem Cache-Miss oder bis zu 0 Taktzyklen bei einem Cache-Hit.

Textur-Speicher (*texture memory*): Ist ein Nur-Lese-Speicher mit Cache optimiert für Textur-Lesezugriffe (sogenannte *Textur-Fetches*). Die Zugriffszeit beträgt 400-600 Taktzyklen bei einem Cache-Miss oder 0 Taktzyklen bei einem Cache-Hit. Der Cache ist optimiert für 2D-Lokalität. Beste Performance wird erreicht, wenn die Threads des Blocks gleichzeitig von Textur-Positionen lesen die dicht beieinander liegen.

3.1.5 Programmbeispiel

In Quellcode 3 ist ein exemplarisches *CUDA*-Programm dargestellt, dass die *CUDA*-API nutzt um Daten auf das Device zu kopieren und den Kernel aus Quellcode 1 auf diesen Daten rechnen lässt.

In Zeile 7 wird Speicher auf dem Host mit Zufallszahlen initialisiert, der dann in Zeile 13 in zuvor allozierten Speicher auf dem Device kopiert wird.

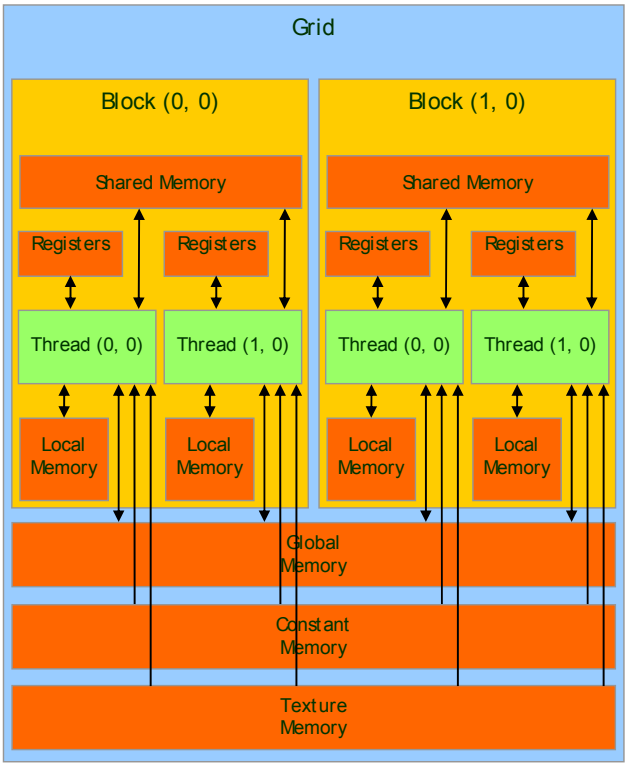

Abbildung 10: Das *CUDA*-Speichermodell. [10]

Danach wird die Ausführungs-Konfiguration des Kernels festgelegt. Die Blockgröße wird in Zeile 16 auf die maximal mögliche Anzahl an Threads gesetzt. Abhängig von der Blockgröße wird in Zeile 17 berechnet, wie viel Blöcke benötigt werden um jedes Element des Eingabe-Arrays mit je einem Thread bearbeiten zu können. Da die Blockgröße sowie das Kernel-Grid eindimensional sind, sind einfache `int`-Variablen (statt `dim3`) ausreichend um die Ausführungs-Konfiguration zu initialisieren. Es folgt dann der eigentlich Aufruf des Kernels. Kernel werden asynchron ausgeführt, d.h. das *CUDA*-Programm fährt unmittelbar nach dem Kernel-Aufruf mit der Ausführung der nächsten Anweisungen fort, unabhängig davon ob der Kernel seine Berechnungen schon fertiggestellt hat oder nicht. Um sicherzustellen, dass vor dem Zugriff auf die Ausgabedaten des Kernels, diese auch vollständig geschrieben wurden, wird in Zeile 22 die Ausführung des *CUDA*-Programms mit `cudaThreadSynchronize()` solange unterbrochen bis alle zuvor aufgerufen Kernel beendet wurden. Danach werden die Ausgabe-Daten zur weiteren Verarbeitung vom Device auf den Host kopiert.

```
1  int main(int argc, const char* argv[])
2  {
3    const int size = 12345, memSize = size * sizeof(float);
4
5    // Speicher auf Host allozieren und initialisieren.
6    float* h_data = (float*)malloc(memSize);
7    for (int i = 0; i < size; ++i) h_data[i] = rand();
8
9    // Speicher auf Device für Ein- und Ausgabedaten allozieren.
10   float* d_in;  cudaMalloc((void**)&d_in,  memSize);
11   float* d_out; cudaMalloc((void**)&d_out, memSize);
12   // Daten von Host auf Device kopieren.
13   cudaMemcpy(d_in, h_data, memSize, cudaMemcpyHostToDevice);
14
15   // Ausführungs-Konfiguration berechnen.
16   int blockSize = 512;
17   int numBlocks = size % blockSize == 0 ?
18                   size / blockSize : size / blockSize + 1;
19
20   // Kernel aufrufen.
21   squareArray<<<numBlocks, blockSize>>>(d_in, d_out, size);
22   cudaThreadSynchronize();
23
24   // Speicher für Ausgabedaten auf Host allozieren.
25   float* h_rslt = (float*)malloc(memSize);
26   // Ausgabedaten vom Device auf Host kopieren.
27   cudaMemcpy(h_rslt, d_out, memSize, cudaMemcpyDeviceToHost);
28
29   return 0;
30 }
```

Quellcode 3: Exemplarisches *CUDA*-Programm, das den Kernel `squareArray()` aus Quellcode 1 aufruft.

3.2 Hardware-Architektur

In den folgenden Abschnitten soll die Hardware-Architektur von *CUDA*-Devices kurz umrissen werden. Da die Implementation dieser Arbeit hauptsächlich auf einer Grafikkarte mit einer GPU vom Typ *G92b* entwickelt und getestet wurde, bezieht sich die Beschreibung auch vorrangig auf diesen GPU-Typ. Im Wesentlichen sind neuere GPUs aber gleich aufgebaut und unterscheiden sich lediglich in der Ausstattung (zum Beispiel mehr Register).

3.2.1 GPU-Architektur

Die GPU eines *CUDA*-Devices verfügt über mehrere *Stream-Multiprozessoren* (*streaming multiprocessors*). Jeder Multiprozessor besteht wiederum aus mehreren *Skalar-Prozessorkernen* (*scalar processor cores*, auch *Thread-Prozessoren* genannt). Des weiteren befindet sich auf jedem Multiprozessor der Shared-Speicher, auf den alle Skalar-Prozessorkerne Zugriff haben. Abbildung 11 veranschaulicht dies am Beispiel einer *G92b*-GPU.

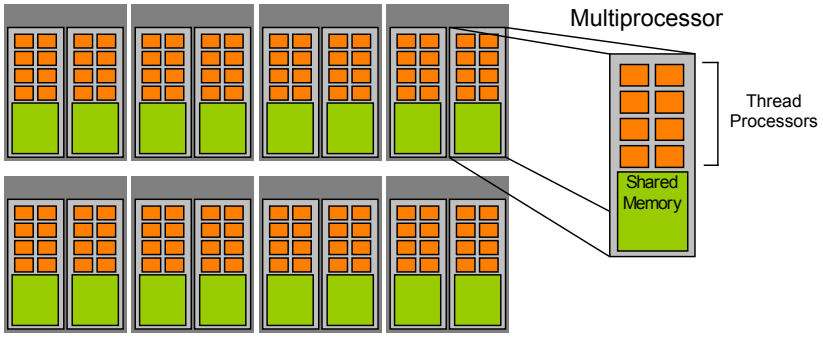

Abbildung 11: Eine *G92b*-GPU besteht aus 16 Multiprozessoren mit jeweils 8 Thread-Prozessoren, verfügt also insgesamt über 128 Thread-Prozessoren. Vgl. [15]

Wird ein Kernel auf dem Device ausgeführt, so werden die Blöcke seines Grids auf die verfügbaren Multiprozessoren der GPU verteilt. Abhängig von den Ressourcen (Shared-Speicher, Register) die ein Block benötigt, können ein oder mehr Blöcke von einem Multiprozessor bearbeitet werden. Die Threads eines Blocks werden nebenläufig auf einen Multiprozessor ausgeführt. Jeder Thread wird dazu einem Thread-Prozessor zugewiesen. Abbildung 12 veranschaulicht diese Zusammenhänge in einer Gegenüberstellung.

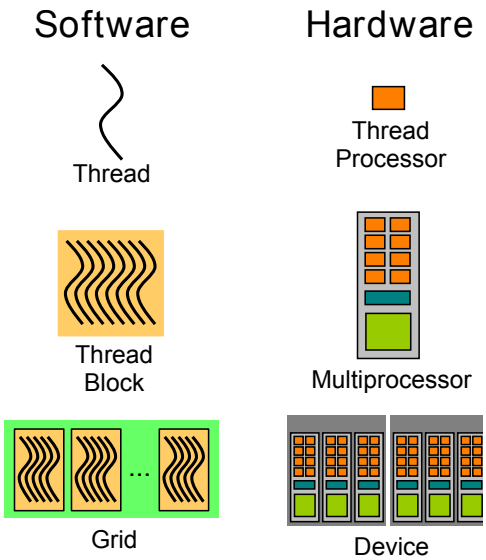

Abbildung 12: Das *CUDA*-Programmiermodell bildet die GPU-Architektur eines *CUDA*-Devices ab. [15]

Wird ein Block von einem Multiprozessor bearbeitet, so teilt er dessen Threads in Gruppen von 32 Threads, sogenannte *Warps* (engl., „Kette“), auf. Die *SIMT-Einheit* (*single-instruction, multiple-thread*) des Multiprozes-

sors bearbeitet dann die Threads eines Warps parallel. Alle Threads des Warps führen dazu gemeinsam einen Befehl des Kernels aus. Kommt es durch Kontrollanweisungen, wie `if`-Konstrukte, zu Verzweigungen im Ausführungspfad des Warps (*branching*), dann werden die unterschiedlichen Befehle serialisiert: Nacheinander werden alle Befehle der Verzweigung von den entsprechenden Threads ausgeführt. Ist der Ausführungspfad für alle Threads nach der Verzweigung wieder gleich, so wird das gesamte Warp wieder parallel ausgeführt. Für maximale Performance sollten Verzweigungen innerhalb eines Warps also vermieden und stattdessen auf zwei verschiedene Warps aufgeteilt werden.

3.2.2 Speicherarchitektur

Die Speicherarchitektur von *CUDA*-Devices ist in Abbildung 13 schematisch dargestellt.

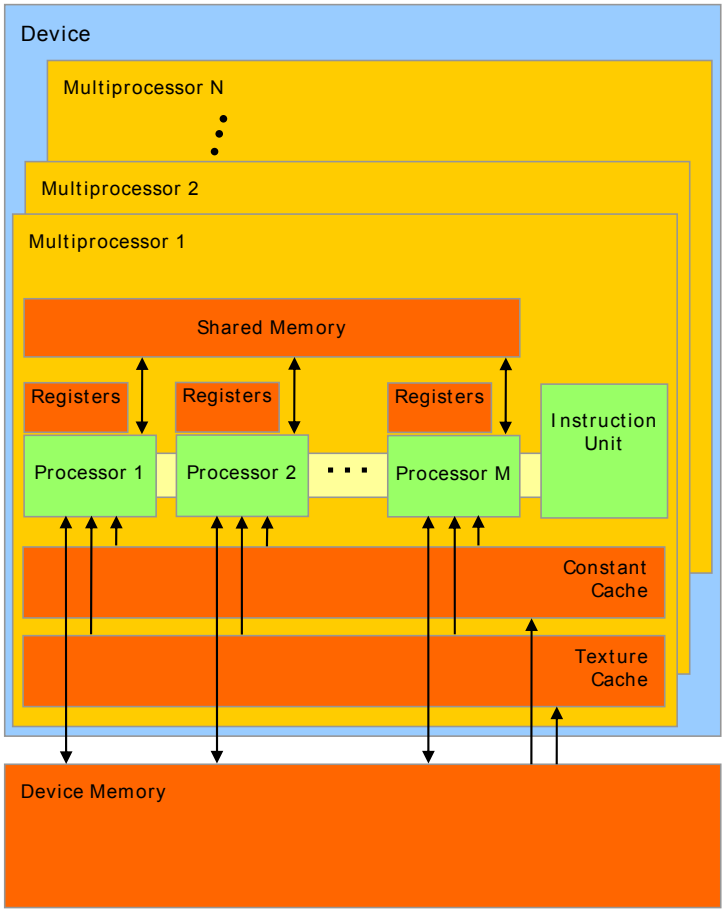

Abbildung 13: Die *CUDA*-Speicherarchitektur. [13]

Direkt auf jedem Multiprozessor (*on-chip*) befinden sich die Register,

der Shared-Speicher sowie die Caches für den konstanten und den Textur-Speicher. Durch die Nähe zu den einzelnen Thread-Prozessoren ergeben sich für diese Speicher sehr kurzen Latenzzeiten. Allerdings sind diese Speicher in ihrer Größe stark beschränkt. So stehen zum Beispiel auf einer *G92b*-GPU pro Multiprozessor nur 16 Kilobyte Shared-Speicher sowie 32 Kilobyte Register-Speicher zur Verfügung.

Der Device-Speicher befindet sich nicht auf der GPU und hat daher eine längere Anbindung und eine entsprechend längere Latenzzeit. Im Gegenzug dazu ist seine Größe nur durch den verfügbaren Grafikspeicher beschränkt. Der Großteil des Device-Speichers wird für den globalen Speicher verwendet.

Der lokale, der konstante sowie der Textur-Speicher sind in Abbildung 13 nicht aufgeführt, da sie physisch keine eigenständigen Speicher sind, sondern reservierte Bereiche des Device-Speichers.

3.3 Performance-Strategien

Um maximale Performance für ein *CUDA*-Programm zu erzielen, sind häufig umfangreiche Code-Optimierungen durchzuführen. Da diese Optimierung das Design der Implementation erheblich beeinflussen, soll hier kurz auf die wichtigsten Optimierungsmöglichkeiten eingegangen werden.

Der Grad der Parallelisierung eines Algorithmus hat den stärksten Einfluss auf die Performance und beeinflusst auch die Skalierbarkeit des Programms. Die Struktur des Algorithmus sollte möglichst feingranular sein, d.h. im Idealfall bearbeitet jeder Thread je ein Datenelement.

Die Geschwindigkeit von Speicher-Zugriffen ist ebenfalls ein Faktor mit großem Einfluss auf die allgemeine Performance eines *CUDA*-Programms. Das zur Bewertung dieser Geschwindigkeit angewandte Maß ist die Rate mit der Daten im Speicher gelesen bzw. geschrieben werden können und wird in Gigabyte pro Sekunde (GB/s) gemessen. Folgende Strategien können helfen die Datenrate eines *CUDA*-Programms drastisch zu erhöhen:

- Minimierung von Datentransfers zwischen Host und Device, da diese sehr langsam sind.
- Je nachdem in welcher Art auf die Daten zugegriffen wird (*access pattern*), sollte der am besten geeignete Speicher genutzt werden. So sollte der globale Speicher für Daten verwendet werden, die blockweise gelesen werden sollen, der Textur-Speicher wenn einzelne Datenelemente gelesen werden, die relativ nah nebeneinander liegen, aber nicht in Blöcken eingelesen werden können und der konstante Speicher wenn man zum Beispiel eine Look-Up-Tabelle implementiert.
- Insbesondere sollte bei Lese- oder Schreibzugriffen auf den globalen Speicher die Fähigkeit des Devices ausgenutzt werden, mehrere Spei-

cherzugriffe zu einen einzigen zusammenzufassen (siehe Abschnitt 3.3.1).

- Gegebenenfalls kann auf einen Datentransfer auch komplett verzichtet werden, indem man die Daten einfach neu berechnet, wenn sie benötigt werden.

Kontrollanweisungen sollten innerhalb von Kerneln mit Bedacht verwendet werden, da sie, insofern sie innerhalb eines Warps die Verzweigung des Ausführungspfades auslösen (*Warp-Divergenz*), die Parallelisierung stören. Warp-Divergenz kann vermieden werden, indem in einem Kernel alle Kontrollstrukturen nur von (`threadIdx / warpSize`) abhängig sind.

3.3.1 „Verschmelzen" von Speicherzugriffen

Ein *CUDA*-Device ist unter bestimmten Umständen in der Lage die Speicherzugriffe von mehreren Threads zu einen einzelnen, großen Speicherzugriff zusammenzufassen. Dies wird als *Speicher-Coalescing* (von „to coalesce", engl., „vereinigen") bezeichnet. Die Speicherzugriffe der 16 Threads der ersten oder der zweiten Hälfte eines Warps werden unter folgenden Bedingungen zu einen Zugriff „verschmolzen":

- Es wird jeweils entweder ein 32-Bit-Word (zum Beispiel ein `float`), ein 64-Bit-Word (`float2`) oder ein 128-Bit-Word (`float4`) gelesen.
- Alle Words müssen sich im gleichen Speichersegment befinden.
- Die Threads müssen in geordneter Reihenfolge auf die Words zugreifen: Der k. Thread der Warp-Hälfte liest bzw. schreibt auch das k. Word im Speichersegment.

Eine typische Vorgehensweise wäre zum Beispiel, dass jeder Thread-Block eines Kernel-Grids unter Ausnutzung von Speicher-Coalescing ein Datensegment in seinen Shared-Speicher kopiert. Alle oder nur ein Teil der Threads eines Blocks bearbeiten dann die gelesenen Daten im Shared-Speicher, wobei von dessen sehr schnellen Zugriffszeiten profitiert wird. Die Ergebnisse der Berechnungen werden zuerst in den Shared-Speicher geschrieben und dann das gesamte Speichersegment wieder gemeinsam von allen Threads per Speicher-Coalescing zurück in den globalen Speicher kopiert.

In Quellcode 4 wird diese Technik exemplarisch angewandt. Das Programm liest blockweise ein 1024×1024 Datenfeld ein, führt Berechnungen auf ihm aus und schreibt dann das Ergebnis in das Datenfeld zurück.

Zu Beginn wird in Zeile 24 ein zweidimensionales Grid mit mehreren 16×16 Blöcken angelegt. Da der Kernel Shared-Speicher verwendet, wird

```
1  __global__
2  void smemKernel(float* data, int width, int height) {
3    extern __shared__ float block[];
4
5    int col = blockIdx.x * blockDim.x + threadIdx.x;
6    int row = blockIdx.y * blockDim.y + threadIdx.y;
7    int dataPos = row * width + col;
8    int blockPos = threadIdx.y * blockDim.x + threadIdx.x;
9
10   block[blockPos] = data[dataPos];
11   __syncthreads();
12
13   // Nutze Shared-Speicher um mit dessen Daten das Ergebnis zu
         berechnen und zu speichern.
14   __syncthreads();
15
16   data[dataPos] = block[blockPos];
17 }
18
19 int main(int argc, char** argv) {
20   const int width = 1024, height = 1024;
21
22   // Initialisiere die Eingabe-Daten.
23
24   dim3 grid, block(16, 16);
25   grid.x = width / block.x;
26   grid.y = height / block.y;
27   int smemSize = block.x * block.y * sizeof(float);
28
29   smemKernel<<<grid, block, smemSize>>>(data, width, height);
30   cudaThreadSynchronize();
31
32   // ...
33 }
```

Quellcode 4: Der Kernel `smemKernel()` nutzt Speicher-Coalescing um mehrere Speicherzugriffe zu einem zu verschmelzen.

in Zeile 27 die Größe des Shared-Speichers für jeden Block in Bytes berechnet, um diese per Ausführungs-Konfiguration den Kernel mitteilen zu können. Im Kernel wird bei der Deklaration des Arrays `block`, welches im Shared-Speicher liegt, mit dem Typ-Qualifizierer `extern` angegeben, dass dessen Größe nicht statisch, sondern dynamisch zur Laufzeit durch die Ausführungs-Konfiguration festgelegt wird. Es folgen die Berechnung globaler Indizes, um zu kontrollieren welcher Thread welches Datenelement aus dem Gesamtdatenfeld liest und an welche Position es im Shared-Speicher geschrieben werden soll. In Zeile 10 geschieht dann der eigentliche, „verschmolzene“ Speicherzugriff: Jeder Thread des Blocks kopiert ein Datenelement vom globalen in den Shared-Speicher. Um sicherzustellen, dass jeder Thread fertig kopiert hat und das Array `block` vollständig gefüllt ist, werden in Zeile 11 mit dem Befehl `__syncthreads()` alle Threads synchronisiert, indem jeder Thread seine Ausführung stoppt bis alle Threads des Blocks diese Anweisung im Kernel erreicht haben. Dann

kann die eigentliche Bearbeitung der Daten erfolgen. In diesem Beispiel wird angenommen, dass jeder Thread abhängig von dem ihn zugeordneten Datenelement ein Ergebnis berechnet und es an Stelle des ursprünglichen Wertes in den Shared-Speicher schreibt. Eine weiteres `__syncthreads()` stellt sicher, dass alle Threads ihre Berechnungen abgeschlossen haben, damit danach die Ergebnisse gemeinsam per Speicher-Coalescing in den globalen Speicher kopiert werden können.

4 Implementation

Für diese Arbeit wurde der *SURF*-Detektor mit *CUDA* für GPUs von *Nvidia*-Grafikkarten implementiert. In den folgenden Abschnitten werden die Struktur und die Arbeitsweise der Implementation detailliert beschrieben.

4.1 Struktur der Implementation

4.1.1 Schnittstelle

Die Schnittstelle der Implementation wurde einfach gehalten: Drei Funktionen und zwei Datenstrukturen reichen aus, um aus einem Bild Positionen von Interessenpunkten zu extrahieren. Die gesamte Schnittstelle ist in Quellcode 5 dargestellt.

Die Datenstruktur `IPoint` (Zeile 1) repräsentiert einen im Bild detektierten Interessenpunkt. Sie besteht aus vier `float`-Werten für die Position, die Skalierung und die Stärke des Interessenpunktes.

Die Funktion `csInitCudaSurf()` (Zeile 17) sowie die Datenstruktur `CsParams` (Zeile 8) dienen der Initialisierung des Devices und der Konfiguration des Detektors. Der Aufruf von `csInitCudaSurf()` ist jedoch nicht zwingend erforderlich. Wird er nicht durchgeführt, benutzt *CUDA* automatisch das Standard-Device des Hosts. Optional kann die zu erwartende Bildgröße übergeben werden, damit der benötigte Speicher im Voraus alloziert werden kann.

Durch den Aufruf von `csSetImage()` (Zeile 21) wird das zu untersuchende Bild auf das Device kopiert und ein Integralbild von diesem erstellt. Das Bild muss ein Grauwertbild mit normierten `float`-Werten im Bereich $[0, 1]$ sein.

Die Funktion `csDetectInterestPoints()` (Zeile 24) startet die eigentliche Detektion von Interessenpunkten. Wie diese im Detail abläuft und implementiert ist wird in den folgenden Abschnitten beschrieben.

Mit `csGetInterestPoints()` (Zeile 26) können die gefundenen Interessenpunkte als ein `IPoint`-Array auf den Host kopiert werden. Die Anzahl der gefundenen Interessenpunkte wird als `unsigned int` zurückgegeben.

```
1  struct IPoint {
2    float x;         // X-Position des Interessenpunktes
3    float y;         // Y-Position
4    float scale;     // Skalierung
5    float strength; // "Interessantheit"
6  };
7
8  struct CsParams {
9    float threshold;              // Schwellwert für Blobs
10   unsigned int octaves;         // Anzahl Oktaven
11   unsigned int intervals;       // Anz. Intervalle pro Oktave
12   unsigned int initSampleStep; // initiales Samplingintervall
13   CsParams(): threshold(0.0004f), octaves(4), intervals(4),
14               initSampleStep(2) {}
15 };
16
17 void csInitCudaSurf(CsParams* p = NULL,
18                     unsigned int imageWidth = 0,
19                     unsigned int imageHeight = 0);
20
21 void csSetImage(float* image, unsigned int width,
22                 unsigned int height);
23
24 void csDetectInterestPoints();
25
26 unsigned int csGetInterestPoints(IPoint** iPoints);
27
28 void csClearCudaSurf();
```

Quellcode 5: Die Schnittstelle der Implementation.

Wurde das Arbeiten mit dem Detektor beendet, sollte jeglicher allozierter Speicher mit `csClearCudaSurf()` (Zeile 28) wieder freigegeben werden.

4.1.2 Dateistruktur

Die oben beschriebene Schnittstelle ist in der *Header*-Datei `cuda_surf.h` definiert. Zum Verwendung der Implementation in einem *C*-Programm muss diese Datei inkludiert werden.

Die *CUDA*-Datei `cuda_surf.cu` ist die Hauptdatei und implementiert diese Schnittstelle mit Hilfe der *CUDA*-API.

Die einzelnen Kernel sind in mehreren *CUH*-Dateien (mit der Endung `.cuh`) definiert und werden von der Hauptdatei inkludiert. Die CUH-Dateien sind nach den Datenstrukturen organisiert, die sie jeweils berechnen:

`integral_image.cuh` implementiert die Kernel zur Berechnung eines Integralbildes.

`blob_map.cuh` implementiert die Kernel zur Erstellung der Skalenraum-Pyramide (genannt *Blob-Map*).

extrema_map.cuh implementiert die Kernel zur Detektion von Extrema im Skalenraum und deren Speichern in der sogenannten *Extrema-Map*.

ipoint_vector.cuh implementiert die Kernel zur Extraktion der Extrema aus der Extrema-Map und deren Speichern in einem Array von Interessenpunkten.

Zusätzlich werden in der Datei `common.cuh` noch gemeinsam genutzte Datenstrukturen und Makros definiert.

4.1.3 Hinweise zu den dargestellten Quellcodes

Zum besseren Verständnis der in den folgenden Abschnitten dargestellten Quellcodes, werden einzelne gemeinsame Elemente hier kurz erläutert.

- Die Präfixe `h_` und `d_` vor Variablennamen zeigen an, dass es sich entweder um eine Variable im Host- (`h`) oder um eine Variable im Device-Speicher (`d`) handelt.
- Der Typ `uint` ist ein `typedef` für den Typen `unsigned int`.
- Die Funktion `iDivUp()` wird verwendet, um bei einer Ganzzahl-Division die nächstgrößere anstatt wie allgemein üblich die nächstkleinere Ganzzahl zu erhalten und ist wie folgt definiert:

```
__device__ __host__
int iDivUp(int a, int b) {
  return (a % b != 0) ? (a / b + 1) : (a / b);
}
```

- Die Datenstruktur `Image` wird verwendet um Bilder kompakt als eine Variable verwalten zu können und ist wie folgt definiert:

```
struct Image {
  float* data;  // Zeiger auf Bilddaten
  uint width;   // Bildbreite
  uint height;  // Bildhöhe
  uint size;    // Bildgröße
  uint memSize; // Speichergröße des Bildes
  Image() : data(NULL), width(0), height(0),
            size(0), memSize(0) {}
};
```

4.2 Berechnung des Integralbildes

Die Berechnung des Integralbildes ist trivial für eine sequentielle CPU-Implementation: Vom Ursprung aus wird über jede Zeile das gesamten Bildes iteriert. Pro Zeile wird jeder Grauwert in einer Zeilen-Summe akkumuliert. Parallel dazu wird über das noch nicht initialisierte Integralbild iteriert und an der jeweils aktuellen Position die Summe aus der bisherigen Zeilen-Summe und dem Wert im Integralbild in der gleichen Spalte, aber eine Zeile darüber gespeichert. In Algorithmus 1 ist dieses Vorgehen als Pseudocode dargestellt.

```
1  FOR row ← 0 TO image height − 1 DO
2    row sum ← 0;
3    FOR column ← 0 TO image width − 1 DO
4      row sum ← row sum + image[row][column];
5      IF row > 0 THEN
6        integral value above ← image[row − 1][column];
7      ELSE
8        integral value above ← 0;
9      END IF
10     integral image[row][column] ← integral value above + row sum;
11   END FOR
12 END FOR
```

Algorithmus 1: Die sequentielle Berechnung eines Integralbildes.

Die Integralbildberechnung lässt sich parallelisieren, indem die gleiche Anzahl an Threads erzeugt wird, wie es Zeilen im Bild gibt. Jeder Thread iteriert dann über eine Zeile und berechnet für jede Pixelposition die Summe aus dem aktuellen Grauwert und allen Grauwerten zuvor. Von jedem Pixel einer Bildzeile wird also die *Präfixsumme* (*prefix sum*, auch *scan* genannt) gebildet und im Integralbild gespeichert. Danach wird das vorläufige Integralbild transponiert und der Vorgang mit dem vorläufigen Integralbild als Eingabe wiederholt. Transponieren der Ausgabe davon ergibt dann das endgültige Integralbild.

Der Kernel in Quellcode 8 implementiert die Berechnung der Präfixsummen in *CUDA*. Allerdings wird dort nicht über die Zeilen, sondern die Spalten iteriert, um auf die Eingabe so zuzugreifen, wie sie auch im Speicher abgelegt ist. Auf diese Weise werden im Speicher nebeneinanderliegende Datenelemente auch von nebeneinanderliegenden Threads bearbeitet und so per Speicher-Coalescing deutlich schneller gelesen und geschrieben.

Der Kernel zum Transponieren eines Bildes orientiert sich stark an dem Kernel `transposeNoBankConflicts()` im Beispielprojekt „transposeNew“ des *CUDA* SDK [14].

Die Funktion `computeIntegralImage()`, dargestellt in Quellcode 9, organisiert den Aufruf der Kernel zum Transponieren und zur Berechnung der Präfixsummen.

```
__global__
void scanColumns(Image inImage, Image outImage) {
  float rowSum = 0.0f;
  uint pos = blockIdx.x * blockDim.x + threadIdx.x;

  if (pos < inImage.width)
    for (uint y = 0; y < inImage.height; ++y) {
      rowSum += inImage.data[pos];
      outImage.data[pos] = rowSum;
      pos += inImage.width;
    }
}
```

Quellcode 8: Der Kernel `scanColumns()` zur Berechnung der Präfixsummen entlang der Spalten eines Bildes.

```
void computeIntegralImage(Image& d_image, Image& d_tempImage,
                          Image& d_integralImage) {
  uint tileDim = 32, blockRows = 8, threads = 512;
  dim3 block(tileDim, blockRows);
  dim3 grid(iDivUp(d_image.width, tileDim),
            iDivUp(d_image.height, tileDim));

  transpose<<<grid, block>>>(d_image, d_integralImage);
  cudaThreadSynchronize();

  scanColumns<<<iDivUp(d_image.width, threads), threads>>>
             (d_integralImage, d_tempImage);
  cudaThreadSynchronize();

  transpose<<<grid, block>>>(d_tempImage, d_integralImage);
  cudaThreadSynchronize();

  scanColumns<<<iDivUp(d_image.height, threads), threads>>>
             (d_integralImage, d_integralImage);
  cudaThreadSynchronize();

  uint pitch = d_integralImage.width * sizeof(float);
  cudaChannelFormatDesc chD = cudaCreateChannelDesc<float>();
  cudaBindTexture2D(NULL, &d_integralImageTex,
                    d_integralImage.data, &chD,
                    d_integralImage.width,
                    d_integralImage.height, pitch);
}
```

Quellcode 9: Die Funktion `computeIntegralImage()` zum Berechnen eines Integralbildes.

Da zum Transponieren des Bildes ein temporärer Zwischenspeicher benötigt wird, hat `computeIntegralImage()` einen zusätzlichen Eingabeparamter `d_tempImage`. Nach der Berechnung der Ausführungs-Konfiguration des `transpose`-Kernels in Zeile 3-6, wird dann das Eingabebild `d_image` transponiert, um danach in Zeile 11 die Präfixsummen der Bildzeilen berechnen zu können. Nach einer weiteren Transposition werden dann in Zeile 18 die Präfixsummen der Spalten berechnet. Nach jedem Kernel-Aufruf wird mit `cudaThreadSynchronize()` sichergestellt, dass alle Daten fertig geschrieben wurden. Abschließend wird das resultierende Integralbild in Zeile 24 an die Textur `d_integralImageTex` gebunden, um einen schnellen Zugriff darauf zu ermöglichen. Da das Integralbild von den LoG-Filtern nicht blockweise, sondern nur einzelne Werte in relativer Nähe zueinander gelesen werden, bietet eine Textur den effizientesten Zugriff.

Zur Vereinfachung wird zusätzlich auch die Device-Funktion `boxIntegral()` (Quellcode 10) definiert, die mit Hilfe der Textur `d_integralImageTex`[2] die Summe aller Grauwerte innerhalb eines gegebenen Rechtecks im Eingabebild berechnet.

```
1  __device__
2  float boxIntegral(int x, int y, int width, int height) {
3    x -= 1; y -= 1;
4
5    float D = tex2D(d_integralImageTex, x + width, y + height);
6    float B = tex2D(d_integralImageTex, x + width, y         );
7    float C = tex2D(d_integralImageTex, x        , y + height);
8    float A = tex2D(d_integralImageTex, x        , y         );
9
10   return D - B - C + A;
11 }
```

Quellcode 10: Die Device-Funktion `boxIntegral()` berechnet mit Hilfe eines Integralbildes Rechtecksummen im Eingabebild.

4.3 Konstruktion des Skalenraumes

Sämtliche Blob-Werte aller Intervalle werden in einen großen Speicherbereich, genannt *Blob-Map*, geschrieben. Dort werden für jede Oktave alle Intervalle hintereinander im Speicher abgelegt. Zur Vereinfachung ist die Blob-Map als ein Struct deklariert:

```
struct BlobMap {
  float* data;         // Device-Zeiger auf Blob-Werte
  uint numIntervals;   // Anzahl der Intervalle
```

[2]Texturen müssen in *CUDA* als globale Variablen angelegt werden. Eine Übergabe an Kernel als Parameter ist nicht möglich. Daher kann in `boxIntegral()` direkt auf `d_integralImageTex` zugegriffen werden.

```
  uint intvlWidth;      // Breite eines Intervalls
  uint intvlHeight;     // Höhe eines Intervalls
  uint intvlSize;       // Größe eines Intervalls
  uint intvlMemSize;    // Speichergröße eines Intervalls
};
```

Neben Attributen wie der Höhe und Breite eines Intervalls (entsprechen der Eingabebildgröße) gibt es in `BlobMap` auch den Zeiger `data`, der auf einen vorher allozierten und mit 0 initialisierten Bereich im Device-Speicher zeigt. Will man nun auf die Blobs eines bestimmten Intervalls zugreifen, so wird aus der Nummer des Intervalls ein Byte-Offset von diesem Zeiger berechnet. Mit

```
uint idx = i * bm.intvlSize + y * bm.intvlWidth + x;
float b = bm.data[idx];
```

erhält man den Blob-Wert `b` an der Position (x, y) im Interval `i` der Blob-Map `bm`.

Die zur Berechnung der Blobs verwendeten LoG-Filter sind in manchen Intervallen gleichgroß: So werden zum Beispiel im zweiten Intervall der ersten Oktave und im ersten Intervall der zweiten Oktave LoG-Filter der Größe 15 genutzt (vgl. Abbildung 5). Diese Intervalle unterscheiden sich demnach lediglich in der Samplingrate mit der Blobs im Bild berechnet werden. In einem solchen Fall muss das Intervall der höheren Oktave also nicht berechnet werden, sondern ergibt sich auch durch Abtasten des korrespondierenden Intervalls aus der niedrigeren Oktave mit halber Samplingrate. Für vier Oktaven mit je vier Intervallen müssen also nicht 16, sondern lediglich 10 Intervalle wirklich berechnet werden. Die Anzahl n der tatsächlich zu berechnenden Intervalle einer Blob-Map, bei einer Konfiguration von o Oktaven und i Intervallen pro Oktave, ergibt sich aus der folgenden Formel:

$$n = i + (i - \text{floor}(i/2))(o - 1) \ . \tag{12}$$

Die Blobs einer Oktave werden für alle Intervalle parallel berechnet. Dazu wird ein Intervall in mehrere 16×16 Blöcke unterteilt. Jeder Thread eines Blocks ist für die Berechnung eines Blob-Wertes zuständig. Wie viele Blöcke pro Intervall benötigt werden, ist abhängig von der Bildgröße und der Samplingrate. Bei einem 512×512 Bild und einem Samplingintervall von 1 werden 32×32 viele Blöcke erzeugt (ohne Randbehandlung). Für 4 Intervalle pro Oktave ergibt sich daraus ein 128×32 großes Grid.

Quellcode 12 zeigt den Kernel, der zur Berechnung der Blobs einer Oktave implementiert wurde. Innerhalb des Kernels berechnet sich jeder Thread zuerst aus dem Index seines Blocks und dem Argument `gridWidthPerIntvl` (gibt die Breite eines Intervalls im Grid in Blöcken an) das Interval für das er den Blob berechnet, um daraus später die Filtergröße errechnen zu können. Danach wird, unter Berücksichtigung des Ab-

stands `border` vom Bildrand, die Position des Grauwertes im Bild berechnet, dessen Blob-Stärke bestimmt werden soll. Nach einer `if`-Abfrage, ob die bestimmte Position innerhalb der zulässigen Grenzen liegt (nötig für Eingabebilder, deren Größe kein Vielfaches der Blockgröße ist), werden die Filtergrößen `l`, `w`, `b` und die Filterfläche `area` berechnet.

```
1  __global__
2  void computeBlobMap(BlobMap blobMap, int gridWidthPerIntvl,
3                      int octave, int intvlOffset, int border,
4                      int sampleStep, int lobeSize) {
5    int interval = blockIdx.x / gridWidthPerIntvl;
6    int x = ((blockIdx.x - interval * gridWidthPerIntvl)
7            * blockDim.x + threadIdx.x) * sampleStep + border;
8    int y = (blockIdx.y * blockDim.y + threadIdx.y)
9            * sampleStep + border;
10
11   if (x < blobMap.intvlWidth && y < blobMap.intvlHeight) {
12     // 1 << (octave + 1) == pow(2, octave + 1)
13     int l = lobeSize + interval * (1 << (octave + 1));
14     int w = l * 3;
15     int b = w / 2;
16     float area = w * w;
17
18     float Dxx = boxIntegral(x-b  , y-l+1, w    , 2*l-1)
19               - boxIntegral(x-l/2, y-l+1, l    , 2*l-1) * 3;
20     float Dyy = boxIntegral(x-l+1, y-b  , 2*l-1, w    )
21               - boxIntegral(x-l+1, y-l/2, 2*l-1, l    ) * 3;
22     float Dxy = boxIntegral(x+1  , y-l  , l    , l    )
23               + boxIntegral(x-l  , y+1  , l    , l    )
24               - boxIntegral(x-l  , y-l  , l    , l    )
25               - boxIntegral(x+1  , y+1  , l    , l    );
26
27     Dxx/=area; Dyy/=area; Dxy/=area;
28
29     float determinant = Dxx * Dyy - 0.91f*0.91f * Dxy * Dxy;
30     int laplacianSign = (Dxx + Dyy >= 0) ? 1 : -1;
31     float blob = (determinant < 0) ?
32                  0.0f : laplacianSign * determinant;
33
34     int pos = (intvlOffset + interval) * blobMap.intvlSize
35               + y * blobMap.intvlWidth + x;
36     blobMap.data[pos] = blob;
37   }
38 }
```

Quellcode 12: Der Kernel `computeBlobMap()` zum parallelem Berechnen der Blob-Werte aller Intervalle einer Oktave.

In Zeile 18-25 werden für jeden LoG-Filter aus den Filtergrößen die einzelnen Teilflächen berechnet und zur Filterantwort aufsummiert. Anschließend werden die einzelnen Filterantworten bezüglich der Filterfläche normiert, um Vergleichbarkeit mit den Antworten von Filtern anderer Größe zu gewährleisten. In Zeile 29 wird nach Gleichung 6 die Determinante der gebildeten Hesse-Matrix berechnet. Das in Zeile 30 berechnete Vorzeichen der Spur der Hesse-Matrix (siehe Gleichung 5) gibt Auskunft über die Art

des Extremums. In Zeile 31 wird überprüft ob ein Extremum vorliegt und der endgültige Blob-Wert berechnet. Das Vorzeichen der Spur wird mit in den Blob-Wert eingerechnet, um später Unterscheiden zu können ob es sich um einen Maximum oder Minimum handelt.

Abschließend wird die Position des Blobs in der Blob-Map bestimmt und der Blob-Wert gespeichert.

Die Berechnung der gesamten Skalenraum-Pyramide übernimmt die Funktion `createBlobMap()`, die in einer Schleife für jede Oktave den Kernel `computeBlobMap()` einmal aufruft und dessen Argumente berechnet. Abschließend wird jedes Intervall der Blob-Map an eine eigene Textur gebunden. Bedingt durch das wachsende Samplingintervall, verhindern die Abstände zwischen den Blob-Werten in der Blob-Map eine effizientes Lesen mit Speicher-Coalescing, weshalb sich Texturen als die schnellere Zugriffsvariante erwiesen.

Zum Lesen eins Blobs aus der Blob-Map wird die Funktion

```
__device__ float getBlob(uint octave, uint interval,
                         float x, float y);
```

bereitgestellt. In dieser wird anhand der übergeben Argumente in einem `switch`-Konstrukt die dem gewünschten Intervall entsprechende Textur ausgewählt und das Texel an der übergebenen Position zurückgeben. Das Verwenden mehrerer Texturen in Verbindung mit einem `switch`-Konstrukt erwies sich, im Gegensatz zum Verwenden einer 3D-Textur, als die effizientere Alternative. Da Texturen in *CUDA* statisch angelegt werden müssen, ist die Anzahl der Oktaven und Intervall nicht völlig variabel. Als Maximum für die Gesamtzahl aller Intervalle n wurde 15 festgelegt. Für eine Konfiguration von o Oktaven und i Intervallen ist also an Hand von Gleichung 12 sicherzustellen, dass dieser Wert nicht überschritten wird.

4.4 Suche lokaler Extrema

Um Extrema zu detektieren untersucht ein Thread die $3 \times 3 \times 3$ Nachbarschaft eines Blobs, um festzustellen ob es sich um den betragsmäßig stärksten Blob, also eine lokales Extremum handelt.

Alle gefunden Extrema werden in einer zweidimensionalen *Extrema-Map* von der Größe des Eingabebildes gespeichert:

```
struct ExtremaMap {
  int* data;
  uint width;
  uint height;
  uint size;
  uint memSize;
};
```

Auf den Speicherbereich der Extrema-Map wird mit dem Attribut `data` zugegriffen. Dieser muss bereits alloziert und mit 0 initialisiert sein. Ein

Wert in der Extrema-Map ungleich 0 weißt auf ein gefundenes Extremum an der korrespondierenden Position im Eingabebild hin.

Da an der gleichen Position im Bild mehrere Interessenpunkte in unterschiedlichen Skalierungsebenen gefunden werden können, ist ein einfaches `int`-Array nicht ausreichend. Verwendet man stattdessen den Vektortyp `int4` kann für vier Oktaven gespeichert werden in welchem Intervall ein Extremum gefunden wurde. Das führt allerdings bei der Auswertung der Extrema-Map dazu, dass das Vierfache an Daten gelesen werden muss. Zudem ist die Extrema-Map in dieser Form lediglich für vier Oktaven verwendbar.

Um das Problem von mehreren Extrema an der selben Position zu lösen ohne die Datenrate zu verringern wurde ein zweidimensionales `int`-Array als Extrema-Map implementiert, dessen 32-Bit-Werte aus kleineren 4-Bit-Werten kodiert werden. Dadurch können in einem `int`-Wert die Extrema einer Skalenraum-Pyramide bestehend aus maximal $8 = \frac{32}{4}$ Oktaven mit jeweils maximal $15_{10} = 1111_2$ Intervallen gespeichert werden (0000_2 ist reserviert um *kein* Extremum anzuzeigen). Abbildung 14 veranschaulicht dies.

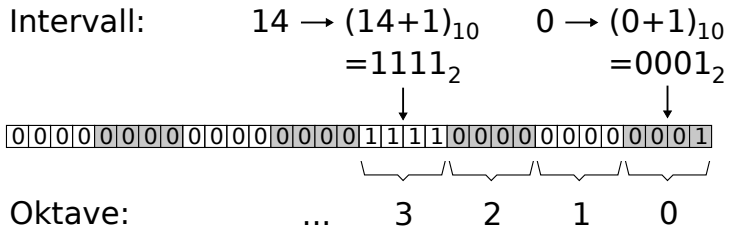

Abbildung 14: Ein 32-Bit-Wert der Extrema-Map wird in acht 4-Bit-Werte aufgeteilt. Pro Bild-Position können so Extrema in maximal acht Oktaven mit bis zu 15 Intervallen detektiert werden. Im Beispiel oben wurden an einer Position zwei Extrema detektiert (in Oktave 0, Interval 0 und Oktave 3, Intervall 14).

Um einen Intervall-Index für eine bestimmte Oktave in ein `int` zu codieren oder aus einem `int` zu extrahieren wurden zwei Funktionen implementiert, die mit Bit-Operationen arbeiten:

```
__device__
void markExtremumAt(int& container, int octave, int interval);

__device__
int hasExtremumAt(int& container, int octave);
```

Der Rückgabewert von `hasExtremumAt()` ist der Index des Intervalls innerhalb der Oktave in der ein Extremum detektiert wurde oder -1 falls sich an der Position keine Extremum befindet.

In Quellcode 14 ist der Kernel `findExtrema()` dargestellt, welcher innerhalb einer Oktave für jeden Blob untersucht, ob es sich um ein Extremum handelt. Dazu berechnet sich jeder Thread des Kernel-Grids zuerst die Position des Blobs, den er untersucht. Zusätzlich wird der Thread auch die Blobs an der gleichen Position in den anderen Intervallen untersuchen. Mit der `for`-Schleife in Zeile 11 iteriert der Thread über alle Intervalle, in denen vollständige $3 \times 3 \times 3$ Nachbarschaft existieren. In dem jeweils ersten und letzten Intervall einer Oktave können also keine Extrema detektiert werden. Nachdem der Blob mit `getBlob()` gelesen wurde, wird in Zeile 15 getestet ob sein Betrag größer als der Schwellwert `threshold` ist und mit der Funktion `isLocalExtremum()` überprüft, ob in seiner Nachbarschaft kein Blob existiert der stärker oder gleichstark ausgeprägt ist. Ist beides der Fall, wird der Wert an der korrespondierenden Position in der Extrema-Map gelesen und das Intervall, in dem das Extremum gefunden wurde, im entsprechenden Bit-Bereich der Oktave gespeichert. Abschließend wird der neu-kodierte Wert wieder in der Extrema-Map abgelegt.

```
1  __global__
2  void findExtrema(ExtremaMap eM, CsParams p, int octave,
3                   int border, int sampleStep) {
4    int x = (blockIdx.x * blockDim.x + threadIdx.x) * sampleStep
5            + border + sampleStep;
6    int y = (blockIdx.y * blockDim.y + threadIdx.y) * sampleStep
7            + border + sampleStep;
8
9    if ((x < eM.width - border - sampleStep)
10       && (y < eM.height - border - sampleStep))
11     for (int interval = 1; interval < p.intervals - 1;
12          ++interval) {
13       float blob = fabsf(getBlob(octave, interval, x, y));
14
15       if (blob > p.threshold && isLocalExtremum(blob, octave,
16                                 interval, x, y, sampleStep)) {
17         int pos = iMul(y, eM.width) + x;
18         int value = extremaMap.data[pos];
19         markExtremumAt(value, octave, interval);
20         eM.data[pos] = value;
21       }
22     }
23 }
```

Quellcode 14: Der Kernel `findExtrema()`.

Die Funktion `createExtremaMap()` ruft diesen Kernel für jeder Oktave einmal auf und berechnet abhängig vom Samplingintervall und der Größe des Randes das passende Kernel-Grid.

Um für jeden Blob eine vollständige $3 \times 3 \times 3$ Nachbarschaft garantieren zu können, darf die Anzahl der Intervalle pro Oktave nicht kleiner als 3 sein.

4.5 Interessenpunkt-Array

Das Umformen der Extrema-Map in ein Array von Interessenpunkten ist auf der GPU nicht trivial. Zum einen können auf der GPU keine dynamisch wachsenden Datenstrukturen verwendet werden, da jeglicher verwendeter Speicher vor der Ausführung von Kerneln bereits alloziert worden sein muss. Zum anderen ist selbst zur Laufzeit nicht die Anzahl der gefundenen Interessenpunkte bekannt, da die Eingabedaten parallel verarbeitet wurden. Um einen Array von Interessenpunkten zu erhalten sind mehrere Arbeitsschritte nötig:

1. Zählen der Interessenpunkte.
    - Um das Potential der GPU voll auszunutzen, sollte parallel mit mehreren Threads gezählt werden. Dies erfordert jedoch das Zusammenzählen der Teilergebnisse der Threads.
2. Host über Anzahl informieren.
    - Da Kernel keine Werte zurückgeben dürfen, muss der Host das Ergebnis per `cudaMemcpy()` aus dem Device-Speicher kopieren.
3. Benötigte Menge Speicher auf dem Device allozieren.
4. Interessenpunkte extrahieren und in das Array einfügen.
    - Auch dieser Schritt sollte für maximale Performance parallelisiert werden.

Cornelis et al. nutzen in ihrer GPU-Implementation des *SURF*-Algorithmus [5] *CUDA*, um in einer dünnbesetzten Matrix, wie zum Beispiel einer Extrema-Map, markierte Positionen zu zählen und in ein Array zu extrahieren. Die in dieser Arbeit implementierte Extraktions-Methode orientiert sich an dem dort vorgestellten Algorithmus.

In einem ersten Pass wird mit dem Kernel `buildIPointVectorPass-1()` (Quellcode 15) über alle Spalten der Extrema-Map iteriert. Zu Beginn wird eine Zählvariable (`numIPoints`) mit 0 initialisiert, die später die Anzahl der Extrema in der durchsuchten Spalte enthalten wird. Dazu wird jeder Wert der Extrema-Map gelesen und in einer Schleife (Zeile 13) mit `hasExtremumAt()` für jede Oktave auf eine detektiertes Extremum hin durchsucht. Ist der zurückgegebene Wert größer als -1, dann befindet sich in der geprüften Oktave ein Extremum und der Zähler `numIPoints` wird um 1 erhöht. Abschließend speichert jeder Thread seinen Extrema-Zähler in dem Array `counters`.

```
1  __global__
2  void buildIPointVectorPass1(CsParams p, ExtremaMap extremaMap,
3                              uint* counters) {
4    uint x = blockIdx.x * blockDim.x + threadIdx.x;
5    uint pos = x;
6    uint numIPoints = 0;
7
8    if (x < extremaMap.width) {
9      for (uint y = 0; y < extremaMap.height; ++y) {
10       int value = extremaMap.data[pos];
11       pos += extremaMap.width;
12
13       for (int octave = 0; octave < p.octaves; ++octave)
14         if (-1 < hasExtremumAt(value, octave))
15           ++numIPoints;
16     }
17
18     counters[x] = numIPoints;
19   }
20 }
```

Quellcode 15: Der Kernel `buildIPointVectorPass1()`.

Im nächsten Schritt werden die einzelnen Zähler in `counters` zur Gesamtzahl an Interessenpunkten aufsummiert. Dies geschieht mit dem Kernel `scanArray()`, der ähnlich dem Kernel `scanColumns()` (Quellcode 8), die Präfixsummen-Funktion implementiert. Nach dem Aufruf von `scanArray()` befindet sich im letzten Element von `counters` die Anzahl aller im Bild detektierten Interessenpunkte. Der Host kopiert diesen Wert anschließend mit `cudaMemcpy()` in seinen Speicher, um das Interessenpunkt-Array zu allozieren.

Im zweiten Pass iteriert der Kernel `buildIPointVectorPass2()` ein weiteres Mal über die Extrema-Map. Diesmal wird ein gefundenes Extremum jedoch als ein `float4` in das Interessenpunkt-Array eingefügt. In der `x`- und `y`-Komponente wird dabei die Position des Interessenpunktes in der Extrema-Map gespeichert. Die Komponenten `z` und `w` des `float4` dienen als Speicher für das Intervall und die Oktave, in dem der Interessenpunkt liegt. Während des Iterierens über die Spalte, dient der korrespondierende Wert in `counters` als ein eindeutiger Index auf das Interessenpunkt-Array. Zu Beginn wird dieser Index aus `counters` gelesen und dann für jedes in der Spalte gefundene Extrema inkrementiert.

4.6 Interpolation der Interessenpunkt-Positionen

Die subpixel-genaue Interpolation der Interessenpunkt-Positionen führt der Kernel `refineIPoints()` durch. Dazu wird für jedes Element im Interessenpunkt-Array ein Thread erzeugt, der diesen Interessenpunkt dann bearbeitet.

Für jeden Interessenpunkt wird zu Beginn das Gleichungssystem 9 be-

stimmt. Die Ableitungen werden dabei durch Differenzen zu den Nachbarpixeln approximiert. Das Inverse der 3×3 Ableitungsmatrix wird explizit berechnet. Der Abstand der interpolierten zur diskreten Position wird dann durch explizites Multiplizieren der Ableitungsmatrix mit dem Ableitungsvektor berechnet. Mit diesem Abstandsvektor wird anschließend die Stabilität des Interessenpunktes überprüft: Ist einer der Komponenten `x` (Δx), `y` (Δy) oder `z` ($\Delta \sigma$) nicht kleiner als $0,5$, so wird der Interessenpunkt für instabil befunden und dessen Position nicht mit dem Abstandsvektor aktualisiert. Stattdessen werden seine `x`-, `y`- und `z`-Komponenten durch Überschreiben mit dem Wert -1 als ungültig markiert. Die endgültige, interpolierte Position eines Interessenpunktes ergibt sich dann nach Gleichung 10, wobei die interpolierte Skalierung $\hat{\sigma}$ nach Gleichung 7 wie folgt aus der Oktave o und dem Intervall i berechnet wird:

$$\hat{\sigma} = \frac{1,2}{9} \cdot 3 \cdot (2^{(o+1)} \cdot (i + \Delta\sigma + 1) + 1) \tag{13}$$

Zusätzlich zur Aktualisierung der Position des Interessenpunktes wird sein Blob-Wert in der `w`-Komponente des `float4` gespeichert. Abschließend wird der aktualisierte (bzw. als instabil markierte) Interessenpunkt in das Interessenpunkt-Array zurückgeschrieben.

Das finale Ergebnis dieser *CUDA*-Implementation des *SURF*-Detektors ist eine `float4`-Array, in dem jedes Element einen Interessenpunkt repräsentiert, wobei die `x`- und `y`-Komponente die subpixel-genau Position des Interessenpunktes im Bild enthalten und in `z` die Skalierung des Interessenpunktes gespeichert ist. Die `w`-Komponente repräsentiert die Stärke („Interessantheit“) des Interessenpunktes. Zusätzlich lässt sich am Vorzeichen dieses Wertes die Art des Interessenpunktes (hell auf dunklem Hintergrund oder dunkel auf hellem Hintergrund) ablesen. Das Interessenpunkt-Array enthält jedoch auch ungültige Interessenpunkte, die als zu instabil befunden wurden. Deren `x`-, `y`- und `z`-Komponente sind jedoch mit dem Wert -1 markiert und lassen sich so leicht herausfiltern.

4.7 Ergebnisse

Der für diese Arbeit implementierte *SURF*-Detektor bearbeitet das in Abbildung 15 dargestellte Bild in durchschnittlich $5,81$ Millisekunden. Die Größe des Bildes beträgt 512×512 Pixel. Die für diesen und alle weiteren Tests in dieser Arbeit verwendete Grafikkarte ist eine *Nvidia GeForce 9800 GTX+* (*G92b* GPU). Die gemessene Laufzeit ist inklusive der benötigten Zeit, um das Bild in den Grafikspeicher zu kopieren ($1,29$ Millisekunden) und exklusive des Zeitbedarfs zum Allozieren des benötigten Grafikspeichers ($37,49$ Millisekunden).

Abbildung 15: Die mit der *SURF*-Implementation dieser Arbeit detektierten Interessenpunkte in einem Beispielbild. Rote Kreise markieren helle Blobs auf dunklem Hintergrund und blaue Kreise dunkle Blobs auf hellem Hintergrund.

Insgesamt wurden in dem Testbild 497 Interessenpunkte detektiert. Davon sind 23 jedoch während des Interpolationsschrittes als zu instabil verworfen worden. Die verbleibenden 474 stabilen Interessenpunkte wurden zur Veranschaulichung in das Bild eingezeichnet. Die grünen Punkte repräsentieren die Positionen der Interessenpunkte und die Kreise um diese deren jeweilige Skalierung. Die Farbe eines Kreises stellt die Art des Interessenpunktes dar: Rot für Maxima-Blobs und Blau für Minima-Blobs.

Um die Korrektheit der detektierten Interessenpunkte überprüfen zu können, wurden die Ergebnisse mit dem Detektor der quelloffenen *SURF*-Implementation *OpenSURF* [6] verglichen.

4.7.1 Vergleich mit OpenSURF: Detektor

Um den Detektor dieser Arbeit mit den Ausgaben von *OpenSURF* vergleichen zu können, wurden beide gleich konfiguriert. Der Skalenraum besteht aus vier Oktaven mit jeweils vier Intervallen und das Samplingintervall beginnt bei 2. Der Schwellwert für Blobs wurde auf $0,4 \cdot 10^{-3}$ gesetzt.

Abbildung 16 zeigt einen direkten Vergleich der Ausgaben beider Detektoren. Die detektierten Interessenpunkte sind in beiden Bildern nahezu identisch. Lediglich die Skalierungen unterscheiden sich im Nachkommabereich, da die GPU des Testsystems keine doppeltgenaue Fließkommaarithmetik beherrscht. Durch die höhere Anzahl an Fließkommaoperationen, die zur Berechnung der Skalierung nach Gleichung 7 nötig sind, schlägt sich dieser Verlust an Genauigkeit dann auf das Endergebnis nieder.

Insgesamt wurden die detektierten Interessenpunkte beider Implementationen in mehreren Testbildern verglichen. Bei gleicher Konfiguration wurden stets gleichviel Interessenpunkte an den gleichen Positionen gefunden.

4.7.2 Vergleich mit OpenSURF: Laufzeit

Zum Vergleichen des Zeitbedarfs des Detektors dieser Arbeit mit dem *OpenSURF*-Detektor wurden beide auf ein Testbild der Größe 2048×2048 zehnmal angewandt und die durchschnittliche Laufzeit berechnet. Um den Einfluss der Bildgröße auf die Laufzeit bewerten zu können, wurde das Testbild schrittweise verkleinert, so dass sieben weitere Testbilder in den Größen 1536×1536, 1280×1280, 1024×1024, 768×768, 640×640, 512×512 und 480×480 vorlagen, und auf diese dann ebenfalls beide Detektoren angewandt während die Laufzeit gemessen wurde.

Für die Laufzeit von *OpenSURF* ist die CPU des Testsystems der wichtigste Faktor. In den Versuchen dieser Arbeit wurde *OpenSURF* auf einem *AMD Athlon 64 X2 Dual Core 3800+* ausgeführt.

Die Ergebnisse der Testreihe sind in Abbildung 17 dargestellt. Für den *OpenSURF*-Detektor ergab sich eine Laufzeit, die die der *CUDA*-Implementation um mehr als eine Zehnerpotenz übersteigt.

Interessanter als ein Vergleich der absoluten Laufzeiten ist jedoch eine Betrachtung des relativen Faktors, um den sich die Laufzeit mit steigender Bildgröße erhöht. Dieser wurde für beide Detektoren ermittelt und ist in Abbildung 18 dargestellt. Der Faktor bezieht sich dabei auf die Laufzeit des jeweiligen Detektors mit einem Eingabebild der Größe 480×480.

Es ist zu erkennen, dass die Laufzeiten beider Detektoren linear mit der Anzahl der Pixel im Bild wachsen. Die Laufzeit des *OpenSURF*-Detektors wächst jedoch stärker. Im Vergleich zum Laufzeitzuwachs des mit *CUDA* implementierten Detektors ergibt sich ein ca. $1,8\times$ stärkerer Anstieg bei *OpenSURF*.

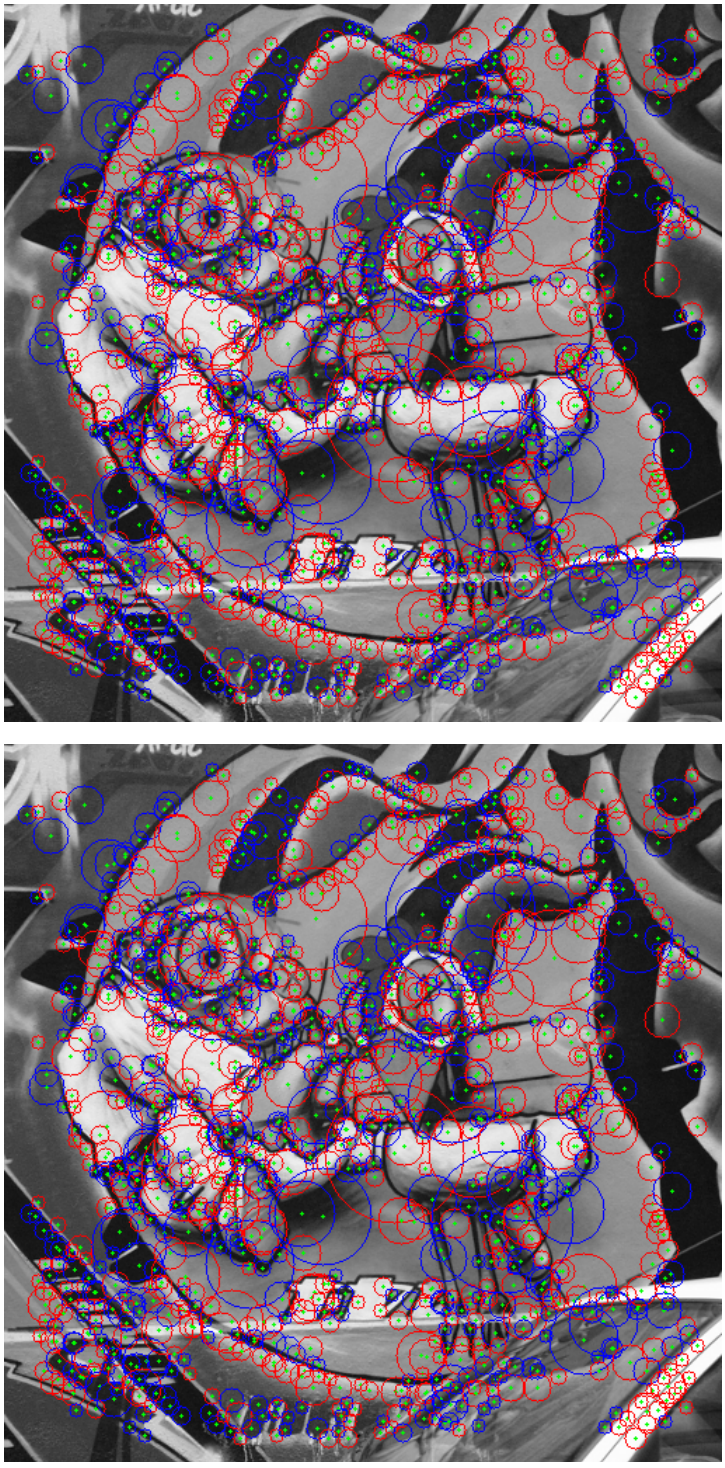

Abbildung 16: Oben: Mit *OpenSURF* detektierte Interessenpunkte. Unten: Ergebnis des SURF-Detektors dieser Arbeit. In beiden Bildern wurden 1079 Interessenpunkte detektiert. Vgl. [11]

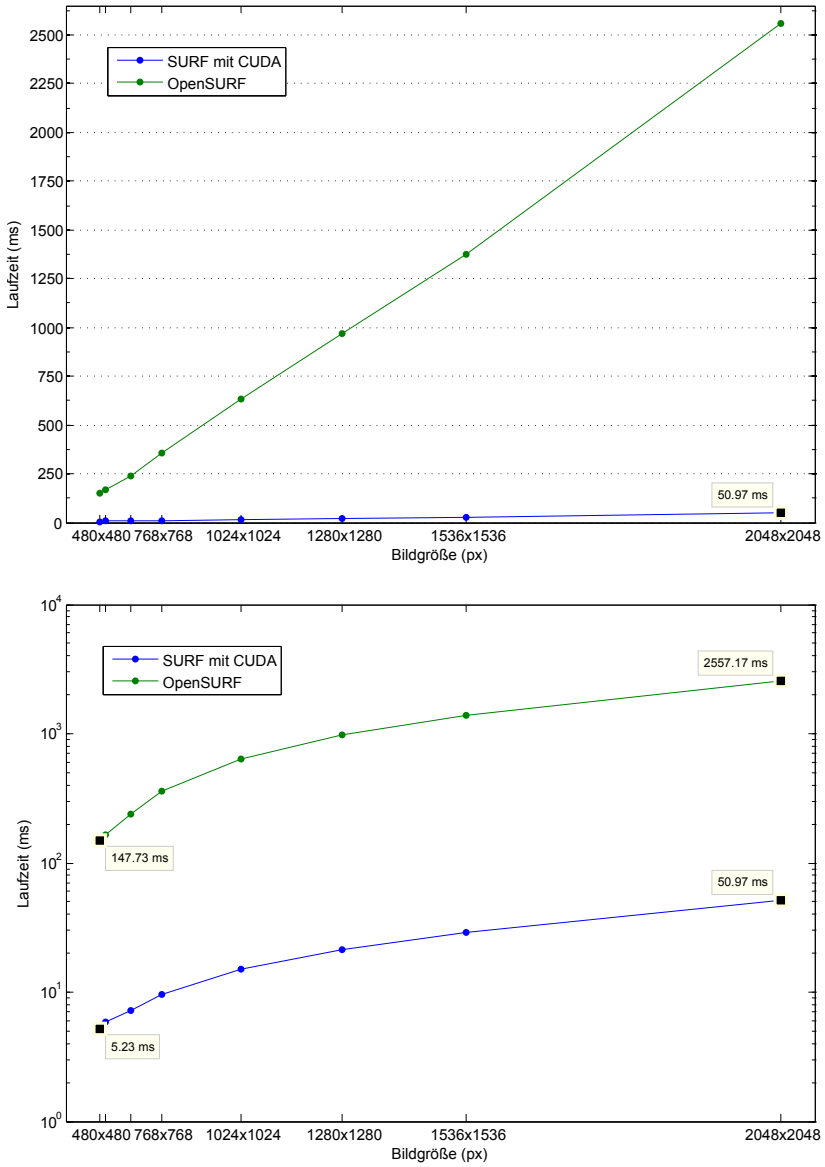

Abbildung 17: Vergleich der Laufzeiten beider Detektor-Implementationen bei unterschiedlichen Bildgrößen. Oben: Lineare Skalierung der Laufzeit. Unten: Logarithmische Skalierung der Laufzeit.

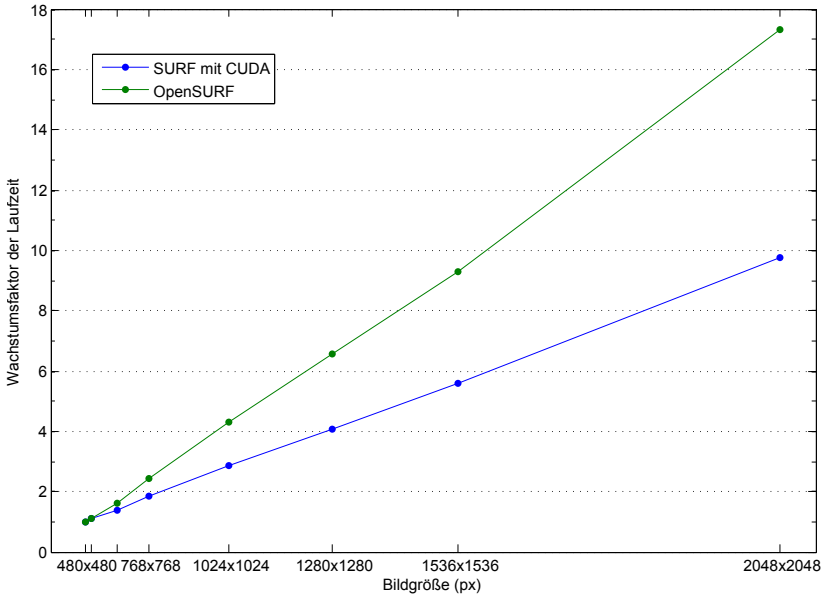

Abbildung 18: Vergleich des Laufzeitverhaltens bei steigenden Bildgrößen. Die Laufzeit des mit *CUDA* implementierten *SURF*-Detektors wächst weniger stark als die des *OpenSURF*-Detektors.

Zusätzlich wurden auch die Laufzeiten der einzelnen Teilschritte der *CUDA*-Implementation des *SURF*-Detektors bei steigenden Bildgrößen gemessen, um Aufschluss darüber zu erhalten welche Berechnungsschritte die Laufzeit dieser Implementation am stärksten beeinflussen. Die Messergebnisse sind in Abbildung 19 dargestellt. Auffällig ist, dass die Laufzeiten der Teilschritte zum Kopieren des Bildes in den Grafikspeicher und zum Berechnen der Blob-Map mit einem wesentlich stärkeren Anstieg wachsen als die restlichen Arbeitsschritte.

5 Fazit

Für diese Arbeit wurde der Interessenpunkt-Detektor des *SURF*-Algorithmus mit *CUDA* implementiert. Die Ausgaben dieser Detektor-Implementation konnten anhand der quelloffenen *SURF*-Implementation *OpenSURF* positiv auf Übereinstimmung hin überprüft werden.

Ziel der Arbeit war es, mit Hilfe von *CUDA* das Parallelisierungspotential von GPUs auszunutzen, um die Detektion von Interessenpunkten in einem Bild zu beschleunigen. Wie in Abschnitt 4.7.2 gezeigt, konnte die Laufzeit gegenüber den für die CPU programmierten *OpenSURF* erheblich verkürzt werden. Zusätzlich war es möglich den Anstieg der Laufzeit bei steigenden Bildgrößen zu verringern.

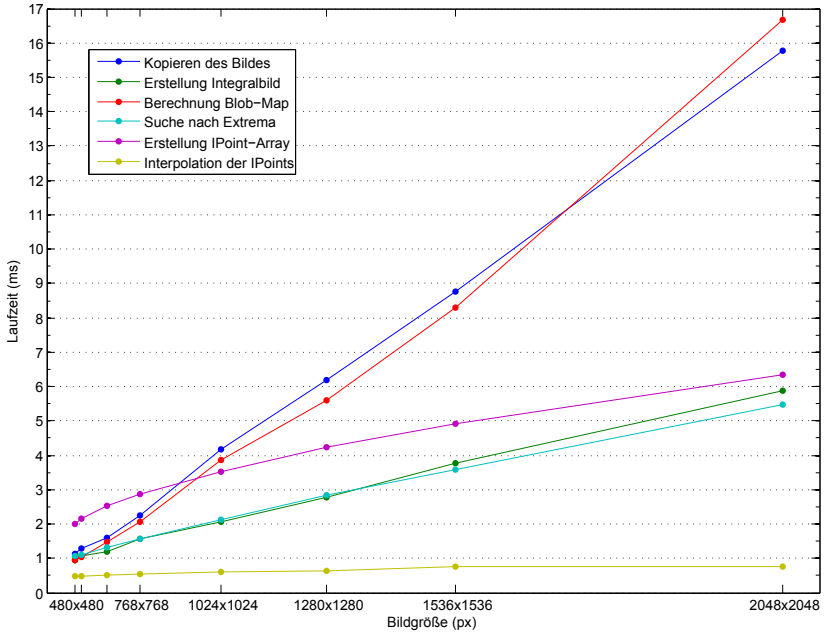

Abbildung 19: kernelRuntime.

5.1 Ausblick

Fortführend ließe sich die Implementation dieser Arbeit auf verschiedene Weisen verbessern. Abbildung 19 offenbart zum Beispiel zwei „Flaschenhälse“ der Applikation: Zum einen das Kopieren des Eingabebildes in den Grafikspeicher und zum anderen die Berechnung der Blob-Map. Der Kopiervorgang an sich kann zwar nicht beschleunigt werden, jedoch könnte man die Möglichkeit von *CUDA* nutzen, das Eingabebild asynchron in den Grafikspeicher zu kopieren. Das bedeutet, dass das *CUDA*-Programm in einem Eingabebild nach Interessenpunkten suchen kann, während parallel dazu bereits das nächste Eingabebild kopiert wird. Solange also genug Berechnungen zur Verfügung stehen, die parallel durchgeführt werden können, würden diese dann den zum Kopieren nötigen Zeitbedarf „überdecken“.

Um die schlechte Skalierung des Kernels zur Berechnung der Blob-Map zu verbessern, könnte der in [17] vorgeschlagene Multipass-Algorithmus zur Bestimmung der Determinante der Hesse-Matrix für *CUDA* abgewandelt und in dieser Implementation verwendet werden. Terriberry et al. konnten mit diesen Verfahren die Anzahl der nötigen Textur-Fetches um bis zu 77 Prozent reduzieren.

5.2 OpenCL

Im Dezember 2008 veröffentlichte die *Khronos Group* die Spezifikation eines neuen Frameworks zur parallelen Programmierung von Multicore-Prozessoren, genannt *OpenCL* (*open computing language*) [8]. Die Spezifikation wurde als eine offener Standard entwickelt, mit dem plattform- und betriebssystemunabhängig die Prozessoren unterschiedlicher Hardware programmiert werden können. Ziel ist es zum einen die GPU-Programme für die Hardware des einen Herstellers auf die - eventuell auch anders geartete - Hardware eines anderen Herstellers portabel zu machen. Und zum anderen sollen in einem *OpenCL*-Programm Berechnungen frei an die verschiedenen Devices im System, die *OpenCL* unterstützen, verteilt werden können. Mit *OpenCL* wird die Abstraktion der unterschiedlichen Prozessortypen, wie CPUs und GPUs, also noch weiter vorangetrieben.

Da inzwischen die zwei größten Hersteller für Grafikkarten, namentlich *Nvidia* und *AMD*, den *OpenCL*-Standard implementiert haben, würde diese Implementation durch eine Portierung nach *OpenCL* von einer höheren Plattformunabhängigkeit profitieren können. Außerdem ließen sich die Berechnung der Blob-Map, sowie die Suche und Extraktion von Interessenpunkt in die einzelnen Oktaven aufspalten und an die verfügbare Hardware verteilen. Bei zwei vergleichbar schnellen Grafikkarten in einem System, wäre so trotz des erhöhten Aufwands durch doppeltes Kopieren und Zusammenfügen der Teilergebnisse ein Performanzgewinn zu erwarten.

Literatur

[1] AMD: *Stream Computing*. – `http://ati.amd.com/technology/streamcomputing/` (Zugriff: 18.10.2009)

[2] BAY, Herbert ; ESS, Andreas ; TUYTELAARS, Tinne ; VAN GOOL, Luc: Speeded-Up Robust Features (SURF). In: *Comput. Vis. Image Underst.* 110 (2008), Nr. 3, S. 346–359. – ISSN 1077–3142

[3] BRADSKI, Gary ; KAEHLER, Adrian: *Learning OpenCV: Computer Vision with the OpenCV Library*. 1st. O'Reilly Media, Inc., 2008. – ISBN 0596516134

[4] BROWN, Matthew ; LOWE, David G.: Invariant Features from Interest Point Groups. In: *British Machine Vision Conference*, 2002, S. 656–665

[5] CORNELIS, Nico ; VAN GOOL, Luc: Fast Scale Invariant Feature Detection and Matching on Programmable Graphics Hardware. In: *Computer Vision and Pattern Recognition Workshop* (2008), S. 1–8. ISBN 978–1–4244–2339–2

[6] EVANS, Christopher: Notes on the OpenSURF Library / University of Bristol. 2009. – Forschungsbericht

[7] HEYMANN, S. ; MÜLLER, K. ; SMOLIC, A. ; FRÖHLICH, B. ; WIEGAND, T.: SIFT Implementation and Optimization for General-Purpose GPU. In: *Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*. Plzen, Tschechische Republik, February 2007, S. 317–322

[8] KHRONOS GROUP: *The OpenCL Specification 1.0.48*. 2008. – `http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf` (Zugriff: 18.10.2009)

[9] LOWE, David G.: Distinctive Image Features from Scale-Invariant Keypoints. In: *Int. J. Comput. Vision* 60 (2004), Nr. 2, S. 91–110. – ISSN 0920–5691

[10] LUEBKE, David: *CUDA Fundamentals*. SIGGRAPH 2008: Beyond Programmable Shading course notes, 14. August 2008. – `http://s08.idav.ucdavis.edu/luebke-cuda-fundamentals.pdf` (Zugriff: 15.10.2009)

[11] MIKOLAJCZYK, Krystian ; SCHMID, Cordelia: *Test Image Dataset for A Performance Evaluation of Local Descriptors*. 2005. – `http://www.robots.ox.ac.uk/~vgg/research/affine/` (Zugriff: 17.10.2009)

[12] NVIDIA: *CUDA*. – `http://www.nvidia.com/object/cuda_home.html` (Zugriff: 18.10.2009)

[13] NVIDIA: *NVIDIA CUDA Programming Guide 2.2.1*. 2009. – `http://developer.download.nvidia.com/compute/cuda/2_21/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.1.pdf` (Zugriff: 15.10.2009)

[14] NVIDIA: *NVIDIA CUDA Software Development Kit 2.2.1*. 2009. – `http://developer.download.nvidia.com/compute/cuda/2_21/sdk/cudasdk_2.21_win_32.exe` (Zugriff: 15.10.2009)

[15] RUETSCH, Greg ; OSTER, Brent: *Getting Started with CUDA*. 2008. – `http://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf` (Zugriff: 15.10.2009)

[16] STANFORD UNIVERSITY GRAPHICS LAB: *BrookGPU*. – `http://graphics.stanford.edu/projects/brookgpu/` (Zugriff: 18.10.2009)

[17] TERRIBERRY, Timothy B. ; FRENCH, Lindley M. ; HELMSEN, John: GPU Accelerating Speeded-Up Robust Features. In: *Proceedings of the 4th International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT'08)*. Atlanta, Georgia, June 2008, S. 355–362

[18] VIOLA, Paul ; JONES, Michael: Rapid Object Detection using a Boosted Cascade of Simple Features. In: *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on* 1 (2001), S. 511. – ISSN 1063–6919

[19] WU, Changchang: *A GPU Implementation of Scale Invariant Feature Transform (SIFT)*. 2007. – `http://cs.unc.edu/~ccwu/siftgpu/` (Zugriff: 18.10.2009)

A Quellcode der Implementation

cuda_surf.h

```
#ifndef _CUDA_SURF_H_
#define _CUDA_SURF_H_

struct CsParams {
  float threshold;
  unsigned int octaves;
  unsigned int intervals;
  unsigned int initSampleStep;
  CsParams() : threshold(0.0004f), octaves(4), intervals(4), initSampleStep(2) {}
};

struct IPoint {
  float x;
  float y;
  float scale;
  float strength;
};

extern "C"
void csInitCudaSurf(CsParams* p = NULL, unsigned int imageWidth = 0, unsigned int imageHeight = 0);

extern "C"
void csSetImage(float* image, unsigned int width, unsigned int height);

extern "C"
void csDetectInterestPoints();

extern "C"
unsigned int csGetInterestPoints(IPoint** iPoints);

extern "C"
void csClearCudaSurf();

#endif // _CUDA_SURF_H_
```

cuda_surf.cu

```
#include <cuda.h>
#include <cutil_inline.h>
#include "common.h"
#include "integral_image.cuh"
#include "blob_map.cuh"
#include "extrema_map.cuh"
#include "ipoint_vector.cuh"

Image h_image;
Image d_image;
Image d_integralImage;
Image d_tempImage;
BlobMap d_blobMap;
ExtremaMap d_extremaMap;
float4* d_iPoints = NULL;
CsParams params;
bool deviceInitialized = false;
uint numberOfIPoints = 0;

void allocGpuMemory(uint width, uint height);

extern "C"
void csInitCudaSurf(CsParams* p = NULL, unsigned int imageWidth = 0, unsigned int imageHeight = 0) {
  if (!deviceInitialized) {
    SC(cudaSetDevice(cutGetMaxGflopsDeviceId()));
    deviceInitialized = true;
  }

  if (p) {
    params.threshold = p->threshold;
    params.initSampleStep = max(p->initSampleStep, 1);
    if (15 >= (p->intervals + (p->intervals - p->intervals / 2) * (p->octaves - 1))) {
      params.octaves = min(max(p->octaves, 1), 7);
      params.intervals = min(max(p->intervals, 3), 15);
    }
  }
}

extern "C"
void csSetImage(float* image, unsigned int width, unsigned int height) {
  allocGpuMemory(width, height);
  h_image.data = image;
  numberOfIPoints = 0;
  if (d_iPoints) {
    SC(cudaFree(d_iPoints));
    d_iPoints = NULL;
  }

  SC(cudaMemcpy(d_image.data, h_image.data, h_image.memSize, cudaMemcpyHostToDevice));
  computeIntegralImage(d_image, d_integralImage, d_tempImage);
```

```
}

extern "C"
void csDetectInterestPoints() {
  createBlobMap(params, d_blobMap);
  createExtremaMap(params, d_extremaMap, d_blobMap);
  numberOfIPoints = createIPointVector(params, d_extremaMap, &d_iPoints);
  if (numberOfIPoints)
    interpolateIPoints(params, d_iPoints, numberOfIPoints);
}

extern "C"
unsigned int csGetInterestPoints(IPoint** iPoints) {
  if (numberOfIPoints) {
    *iPoints = (IPoint*)malloc(numberOfIPoints * sizeof(IPoint));
    SC(cudaMemcpy(*iPoints, d_iPoints, numberOfIPoints * sizeof(float4), cudaMemcpyDeviceToHost));
  }
  return numberOfIPoints;
}

extern "C"
void csClearCudaSurf() {
  SC(cudaFree(d_image.data));
  SC(cudaFree(d_integralImage.data));
  SC(cudaUnbindTexture(d_integralImageTex));
  SC(cudaFree(d_tempImage.data));
  freeBlobMap(d_blobMap);
  freeExtremaMap(d_extremaMap);
}

void allocGpuMemory(uint width, uint height) {
  if (width == h_image.width && height == h_image.height)
    return;

  uint oldMemSize = h_image.memSize;

  h_image.width   = d_image.width   = width;
  h_image.height  = d_image.height  = height;
  h_image.size    = d_image.size    = width * height;
  h_image.memSize = d_image.memSize = h_image.size * sizeof(float);
  d_integralImage.width   = d_tempImage.width   = h_image.width;
  d_integralImage.height  = d_tempImage.height  = h_image.height;
  d_integralImage.size    = d_tempImage.size    = h_image.size;
  d_integralImage.memSize = d_tempImage.memSize = h_image.memSize;

  if (h_image.memSize > oldMemSize) {
    SC(cudaMalloc((void**)&d_image.data, d_image.memSize));
    SC(cudaMalloc((void**)&d_integralImage.data, d_integralImage.memSize));
    SC(cudaMalloc((void**)&d_tempImage.data, d_tempImage.memSize));
    allocateBlobMap(d_blobMap, h_image, params);
    allocateExtremaMap(d_extremaMap, h_image);
  }
}
```

common.h

```
#ifndef _COMMON_H_
#define _COMMON_H_

#define SC(cudaCall)   cutilSafeCall(cudaCall)
#define CE(cutilCall)  cutilCheckError(cutilCall)
#define CK()           cutilCheckMsg("kernel invocation")
#define SM(mallocCall) cutilSafeMalloc(mallocCall)

typedef unsigned int uint;

struct CsParams {
  float threshold;
  unsigned int octaves;
  unsigned int intervals;
  unsigned int initSampleStep;
  CsParams() : threshold(0.0004f), octaves(4), intervals(4), initSampleStep(2) {}
};

struct IPoint {
  float x;
  float y;
  float scale;
  float strength;
};

struct Image {
  float* data;
  uint width;
  uint height;
  uint size;
  uint memSize;
  Image() : data(NULL), width(0), height(0), size(0), memSize(0) {}
};

struct Matrix3x3 {
  float c11, c12, c13;
  float c21, c22, c23;
  float c31, c32, c33;
};
```

```
__device__ __host__
int iDivUp(int a, int b) { return (a % b != 0) ? (a / b + 1) : (a / b); }

/* Fast integer multiplication with 24-bit precision. */
__device__
int iMul(int a, int b) { return __mul24(a, b); }

/* Faster integer multiplication for factors which are a power of two.
 *   iMulS(a, b) == a << b == a * pow(2, b)
 */
__device__ __host__
int iMulS(int a, uint powerOfTwo) { return (a << powerOfTwo); }

/* Fast integer division for divisors which are a power of two.
 *   iDivS(a, b) == a >> b == a / pow(2, b)
 */
__device__ __host__
int iDivS(int a, uint powerOfTwo) { return (a >> powerOfTwo); }

#endif // _COMMON_H_
```

integral_image.cuh

```
#ifndef _INTEGRAL_IMAGE_CUH_
#define _INTEGRAL_IMAGE_CUH_

#include "common.h"

texture<float, 2> d_integralImageTex;

__device__
float boxIntegral(int x, int y, uint regionWidth, uint regionHeight) {
  x -= 1;
  y -= 1;

  const float ABCD = tex2D(d_integralImageTex, x + regionWidth, y + regionHeight);
  const float AB   = tex2D(d_integralImageTex, x + regionWidth, y               );
  const float AC   = tex2D(d_integralImageTex, x,               y + regionHeight);
  const float A    = tex2D(d_integralImageTex, x,               y               );
  return ABCD - AB - AC + A;
}

__global__
void scanColumns(Image inImage, Image outImage) {
  float sum = 0.0f;
  uint pos = iMulS(blockIdx.x, 9) + threadIdx.x;

  if (pos < inImage.width)
    for (uint y = 0; y < inImage.height; ++y) {
      sum += inImage.data[pos];
      outImage.data[pos] = sum;
      pos += inImage.width;
    }
}

__global__
void transpose(Image inImage, Image outImage) {
  const uint tileDim = 32, blockRows = 8;
  __shared__ float tile[tileDim][tileDim + 1];

  uint xPos = iMul(blockIdx.x, tileDim) + threadIdx.x;
  uint yPos = iMul(blockIdx.y, tileDim) + threadIdx.y;
  uint inPos = iMul(yPos, inImage.width) + xPos;

  if (xPos < inImage.width)
    for (uint i = 0; i < tileDim; i += blockRows)
      if (yPos + i < inImage.height)
        tile[threadIdx.y + i][threadIdx.x] = inImage.data[inPos + iMul(i, inImage.width)];

  __syncthreads();

  xPos = iMul(blockIdx.y, tileDim) + threadIdx.x;
  yPos = iMul(blockIdx.x, tileDim) + threadIdx.y;
  uint outPos = iMul(yPos, inImage.height) + xPos;

  if (xPos < inImage.height)
    for (uint i = 0; i < tileDim; i += blockRows)
      if (yPos + i < inImage.width)
        outImage.data[outPos + iMul(i, inImage.height)] = tile[threadIdx.x][threadIdx.y + i];
}

void computeIntegralImage(Image& d_image, Image& d_integralImage, Image& d_tempImage) {
  const uint tileDim = 32, blockRows = 8, numThreads = 512;
  dim3 blockDim(tileDim, blockRows);
  dim3 gridDim(iDivUp(d_image.width, tileDim), iDivUp(d_image.height, tileDim));

  /* Transpose so rows become columns. */
  transpose<<<gridDim, blockDim>>>(d_image, d_integralImage);
  SC(cudaThreadSynchronize());
  CK();

  /* Scan (sum up) image rows. */
  scanColumns<<<iDivUp(d_image.width, numThreads), numThreads>>>(d_integralImage, d_tempImage);
  SC(cudaThreadSynchronize());
```

```
  CK();

  /* Transpose back. */
  transpose<<<gridDim, blockDim>>>(d_tempImage, d_integralImage);
  SC(cudaThreadSynchronize());
  CK();

  /* Scan image columns in place. */
  scanColumns<<<iDivUp(d_image.height, numThreads), numThreads>>>(d_integralImage, d_integralImage);
  SC(cudaThreadSynchronize());
  CK();

  /* Bind integral image to a texture. */
  size_t pitch = d_integralImage.width * sizeof(float);
  const cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
  SC(cudaBindTexture2D(NULL, &d_integralImageTex, d_integralImage.data, &channelDesc,
                       d_integralImage.width, d_integralImage.height, pitch));
}

#endif // _INTEGRAL_IMAGE_CUH_
```

blob_map.cuh

```
#ifndef _BLOB_MAP_CUH_
#define _BLOB_MAP_CUH_

#include "common.h"
#include "integral_image.cuh"

struct BlobMap {
  float* data;
  uint numIntvls;
  uint intvlWidth;
  uint intvlHeight;
  uint intvlSize;
  uint intvlMemSize;
};

texture<float, 2> d_blobMapTex00, d_blobMapTex01, d_blobMapTex02, d_blobMapTex03, d_blobMapTex04,
                  d_blobMapTex05, d_blobMapTex06, d_blobMapTex07, d_blobMapTex08, d_blobMapTex09,
                  d_blobMapTex10, d_blobMapTex11, d_blobMapTex12, d_blobMapTex13, d_blobMapTex14;

__device__
uint getAbsIntervalIndex(uint octave, uint interval) {
  switch (octave) {
  case 0:
    return interval;
  default:
    switch (interval) {
    case 0: return iMulS(octave, 1) - 1;
    default: return iMulS(octave, 1) + interval;
    }
  }
}

__device__ __noinline__
float getBlob(uint octave, uint interval, float x, float y) {
  switch (getAbsIntervalIndex(octave, interval)) {
  case  0: return tex2D(d_blobMapTex00, x, y);
  case  1: return tex2D(d_blobMapTex01, x, y);
  case  2: return tex2D(d_blobMapTex02, x, y);
  case  3: return tex2D(d_blobMapTex03, x, y);
  case  4: return tex2D(d_blobMapTex04, x, y);
  case  5: return tex2D(d_blobMapTex05, x, y);
  case  6: return tex2D(d_blobMapTex06, x, y);
  case  7: return tex2D(d_blobMapTex07, x, y);
  case  8: return tex2D(d_blobMapTex08, x, y);
  case  9: return tex2D(d_blobMapTex09, x, y);
  case 10: return tex2D(d_blobMapTex10, x, y);
  case 11: return tex2D(d_blobMapTex11, x, y);
  case 12: return tex2D(d_blobMapTex12, x, y);
  case 13: return tex2D(d_blobMapTex13, x, y);
  case 14: return tex2D(d_blobMapTex14, x, y);
  default: return 0.0f;
  }
}

__device__
float getAbsBlob(uint octave, uint interval, float x, float y) {
  return fabsf(getBlob(octave, interval, x, y));
}

void allocateBlobMap(BlobMap& b, Image& i, CsParams& p) {
  b.numIntvls = p.intervals + (p.intervals - p.intervals / 2) * (p.octaves - 1);
  b.intvlWidth = i.width;
  b.intvlHeight = i.height;
  b.intvlSize = i.size;
  b.intvlMemSize = i.memSize;

  SC(cudaMalloc((void**)&b.data, b.numIntvls * b.intvlMemSize));
  SC(cudaMemset(b.data, 0, b.numIntvls * b.intvlMemSize));
}

__global__
void computeBlobMap(BlobMap blobMap, int gridWidthPerIntvl, int octave, int intvlOffset, int border, int sampleStep,
       int lobeSize) {
```

```
  int interval = blockIdx.x / gridWidthPerIntvl;
  int x = iMul(iMul(blockIdx.x - iMul(interval, gridWidthPerIntvl), blockDim.x) + threadIdx.x, sampleStep) + border;
  int y = iMul(iMul(blockIdx.y, blockDim.y) + threadIdx.y, sampleStep) + border;

  if (x < blobMap.intvlWidth - border && y < blobMap.intvlHeight - border) {
    int l = lobeSize + iMul(interval, iMulS(2, octave));
    int w = iMul(l, 3);
    int b = iDivS(w, 1);
    float invArea = 1.0f / iMul(w, w);

    float Dxx = boxIntegral(x - b,           y - l + 1,       w,               iMulS(l, 1) - 1)
              - boxIntegral(x - iDivS(l, 1), y - l + 1,       l,               iMulS(l, 1) - 1) * 3;
    float Dyy = boxIntegral(x - l + 1,       y - b,           iMulS(l, 1) - 1, w             )
              - boxIntegral(x - l + 1,       y - iDivS(l, 1), iMulS(l, 1) - 1, l             ) * 3;
    float Dxy = boxIntegral(x + 1,           y - l,           l,               l             )
              + boxIntegral(x - l,           y + 1,           l,               l             )
              - boxIntegral(x - l,           y - l,           l,               l             )
              - boxIntegral(x + 1,           y + 1,           l,               l             );

    Dxx *= invArea;
    Dyy *= invArea;
    Dxy *= invArea;

    float determinant = Dxx * Dyy - 0.81f * Dxy * Dxy;
    int laplacianSign = (Dxx + Dyy >= 0) ? 1 : -1;
    float blob = (determinant < 0) ? 0 : laplacianSign * determinant;

    int pos = iMul(intvlOffset + interval, blobMap.intvlSize) + iMul(y, blobMap.intvlWidth) + x;
    blobMap.data[pos] = blob;
  }
}

void createBlobMap(CsParams& params, BlobMap& blobMap) {
  dim3 blockDim(16, 16);
  int octave = 0;
  int intvlOffset = 0;
  int sampleStep = params.initSampleStep;
  int lobeSize = 3;
  dim3 gridDim;
  int border = 3 * (iMulS(2, octave) * params.intervals + 1) / 2 + 1;
  int gridWidthPerIntvl = iDivUp(iDivUp(blobMap.intvlWidth - border * 2, blockDim.x), sampleStep);
  gridDim.x = gridWidthPerIntvl * params.intervals;
  gridDim.y = iDivUp(iDivUp(blobMap.intvlHeight - border * 2, blockDim.y), sampleStep);

  /* Compute blob responses for first octave. */
  computeBlobMap<<<gridDim, blockDim>>>(blobMap, gridWidthPerIntvl, octave, intvlOffset, border, sampleStep, lobeSize);

  int uniqIntvlsPerOctv = params.intervals - params.intervals / 2;
  lobeSize = 13;
  intvlOffset = params.intervals;

  /* Compute blob responses for remaining octaves. */
  for (octave = 1; octave < params.octaves; ++octave)
  {
    sampleStep *= 2;
    border = (border - 1) * 2;
    gridWidthPerIntvl = iDivUp(iDivUp(blobMap.intvlWidth - border * 2, blockDim.x), sampleStep);
    gridDim.x = gridWidthPerIntvl * uniqIntvlsPerOctv;
    gridDim.y = iDivUp(iDivUp(blobMap.intvlHeight - border * 2, blockDim.y), sampleStep);

    computeBlobMap<<<gridDim, blockDim>>>(blobMap, gridWidthPerIntvl, octave, intvlOffset, border, sampleStep, lobeSize
          );

    lobeSize = lobeSize * 2 - 1;
    intvlOffset += 2;
  }

  SC(cudaThreadSynchronize());
  CK();

  size_t pitch = blobMap.intvlWidth * sizeof(float);
  const cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
  int intvlsToBind = blobMap.numIntvls;
  float* data = blobMap.data;

  if (intvlsToBind)
    SC(cudaBindTexture2D(NULL, &d_blobMapTex00, data, &channelDesc,
                         blobMap.intvlWidth, blobMap.intvlHeight, pitch));
  else return;

  --intvlsToBind;
  data += blobMap.intvlSize;
  if (intvlsToBind)
    SC(cudaBindTexture2D(NULL, &d_blobMapTex01, data, &channelDesc,
                         blobMap.intvlWidth, blobMap.intvlHeight, pitch));
  else return;

  --intvlsToBind;
  data += blobMap.intvlSize;
  if (intvlsToBind)
    SC(cudaBindTexture2D(NULL, &d_blobMapTex02, data, &channelDesc,
                         blobMap.intvlWidth, blobMap.intvlHeight, pitch));
  else return;

  --intvlsToBind;
  data += blobMap.intvlSize;
  if (intvlsToBind)
    SC(cudaBindTexture2D(NULL, &d_blobMapTex03, data, &channelDesc,
```

```
                        blobMap.intvlWidth, blobMap.intvlHeight, pitch));
  else return;

  --intvlsToBind;
  data += blobMap.intvlSize;
  if (intvlsToBind)
    SC(cudaBindTexture2D(NULL, &d_blobMapTex04, data, &channelDesc,
                        blobMap.intvlWidth, blobMap.intvlHeight, pitch));
  else return;

  --intvlsToBind;
  data += blobMap.intvlSize;
  if (intvlsToBind)
    SC(cudaBindTexture2D(NULL, &d_blobMapTex05, data, &channelDesc,
                        blobMap.intvlWidth, blobMap.intvlHeight, pitch));
  else return;

  --intvlsToBind;
  data += blobMap.intvlSize;
  if (intvlsToBind)
    SC(cudaBindTexture2D(NULL, &d_blobMapTex06, data, &channelDesc,
                        blobMap.intvlWidth, blobMap.intvlHeight, pitch));
  else return;

  --intvlsToBind;
  data += blobMap.intvlSize;
  if (intvlsToBind)
    SC(cudaBindTexture2D(NULL, &d_blobMapTex07, data, &channelDesc,
                        blobMap.intvlWidth, blobMap.intvlHeight, pitch));
  else return;

  --intvlsToBind;
  data += blobMap.intvlSize;
  if (intvlsToBind)
    SC(cudaBindTexture2D(NULL, &d_blobMapTex08, data, &channelDesc,
                        blobMap.intvlWidth, blobMap.intvlHeight, pitch));
  else return;

  --intvlsToBind;
  data += blobMap.intvlSize;
  if (intvlsToBind)
    SC(cudaBindTexture2D(NULL, &d_blobMapTex09, data, &channelDesc,
                        blobMap.intvlWidth, blobMap.intvlHeight, pitch));
  else return;

  --intvlsToBind;
  data += blobMap.intvlSize;
  if (intvlsToBind)
    SC(cudaBindTexture2D(NULL, &d_blobMapTex10, data, &channelDesc,
                        blobMap.intvlWidth, blobMap.intvlHeight, pitch));
  else return;

  --intvlsToBind;
  data += blobMap.intvlSize;
  if (intvlsToBind)
    SC(cudaBindTexture2D(NULL, &d_blobMapTex11, data, &channelDesc,
                        blobMap.intvlWidth, blobMap.intvlHeight, pitch));
  else return;

  --intvlsToBind;
  data += blobMap.intvlSize;
  if (intvlsToBind)
    SC(cudaBindTexture2D(NULL, &d_blobMapTex12, data, &channelDesc,
                        blobMap.intvlWidth, blobMap.intvlHeight, pitch));
  else return;

  --intvlsToBind;
  data += blobMap.intvlSize;
  if (intvlsToBind)
    SC(cudaBindTexture2D(NULL, &d_blobMapTex13, data, &channelDesc,
                        blobMap.intvlWidth, blobMap.intvlHeight, pitch));
  else return;

  --intvlsToBind;
  data += blobMap.intvlSize;
  if (intvlsToBind)
    SC(cudaBindTexture2D(NULL, &d_blobMapTex14, data, &channelDesc,
                        blobMap.intvlWidth, blobMap.intvlHeight, pitch));
}

void freeBlobMap(BlobMap& blobMap) {
  SC(cudaUnbindTexture(d_blobMapTex00)); SC(cudaUnbindTexture(d_blobMapTex01));
  SC(cudaUnbindTexture(d_blobMapTex02)); SC(cudaUnbindTexture(d_blobMapTex03));
  SC(cudaUnbindTexture(d_blobMapTex04)); SC(cudaUnbindTexture(d_blobMapTex05));
  SC(cudaUnbindTexture(d_blobMapTex06)); SC(cudaUnbindTexture(d_blobMapTex07));
  SC(cudaUnbindTexture(d_blobMapTex08)); SC(cudaUnbindTexture(d_blobMapTex09));
  SC(cudaUnbindTexture(d_blobMapTex10)); SC(cudaUnbindTexture(d_blobMapTex11));
  SC(cudaUnbindTexture(d_blobMapTex12)); SC(cudaUnbindTexture(d_blobMapTex13));
  SC(cudaUnbindTexture(d_blobMapTex14));
  SC(cudaFree(blobMap.data));
}

#endif // _BLOB_MAP_CUH_
```

extrema_map.cuh

```
#ifndef _EXTREMA_MAP_CUH_
#define _EXTREMA_MAP_CUH_

#include "common.h"
#include "blob_map.cuh"

struct ExtremaMap {
  uint* data;
  uint width;
  uint height;
  uint size;
  uint memSize;
};

void allocateExtremaMap(ExtremaMap& extremaMap, Image& image) {
  extremaMap.width   = image.width;
  extremaMap.height  = image.height;
  extremaMap.size    = image.size;
  extremaMap.memSize = image.size * sizeof(uint);

  SC(cudaMalloc((void**)&extremaMap.data, extremaMap.memSize));
  SC(cudaMemset(extremaMap.data, 0, extremaMap.memSize));
}

__device__
void markExtremumAt(int& container, int octave, int interval) {
  /* Create shifted bit mask of the value to store. */
  int valueMask = ((interval + 1) << iMulS(octave, 2));
  /* Store value in container. */
  container |= valueMask;
}

__device__
int hasExtremumAt(int& container, int octave) {
  return ((container >> iMulS(octave, 2)) & 15) - 1;
}

__device__
bool isLocalExtremum(float v, int o, int i, int x, int y, int s) {
  if (v > getAbsBlob(o,i-1,x-s,y-s) && v > getAbsBlob(o,i-1,x  ,y-s) && v > getAbsBlob(o,i-1,x+s,y-s)
   && v > getAbsBlob(o,i-1,x-s,y  ) && v > getAbsBlob(o,i-1,x  ,y  ) && v > getAbsBlob(o,i-1,x+s,y  )
   && v > getAbsBlob(o,i-1,x-s,y+s) && v > getAbsBlob(o,i-1,x  ,y+s) && v > getAbsBlob(o,i-1,x+s,y+s)

   && v > getAbsBlob(o,i  ,x-s,y-s) && v > getAbsBlob(o,i  ,x  ,y-s) && v > getAbsBlob(o,i  ,x+s,y-s)
   && v > getAbsBlob(o,i  ,x-s,y  )                                  && v > getAbsBlob(o,i  ,x+s,y  )
   && v > getAbsBlob(o,i  ,x-s,y+s) && v > getAbsBlob(o,i  ,x  ,y+s) && v > getAbsBlob(o,i  ,x+s,y+s)

   && v > getAbsBlob(o,i+1,x-s,y-s) && v > getAbsBlob(o,i+1,x  ,y-s) && v > getAbsBlob(o,i+1,x+s,y-s)
   && v > getAbsBlob(o,i+1,x-s,y  ) && v > getAbsBlob(o,i+1,x  ,y  ) && v > getAbsBlob(o,i+1,x+s,y  )
   && v > getAbsBlob(o,i+1,x-s,y+s) && v > getAbsBlob(o,i+1,x  ,y+s) && v > getAbsBlob(o,i+1,x+s,y+s))
  {
    return true;
  }

  return false;
}

__global__
void findExtrema(CsParams p, ExtremaMap extremaMap, int octave, int border, int sampleStep) {
  int x = iMul(iMul(blockIdx.x, blockDim.x) + threadIdx.x, sampleStep) + border + sampleStep;
  int y = iMul(iMul(blockIdx.y, blockDim.y) + threadIdx.y, sampleStep) + border + sampleStep;

  if ((x < extremaMap.width - border - sampleStep)
      && (y < extremaMap.height - border - sampleStep))
    for (int interval = 1; interval < p.intervals - 1; ++interval) {
      float blob = getAbsBlob(octave, interval, x, y);

      if (blob > p.threshold && isLocalExtremum(blob, octave, interval, x, y, sampleStep)) {
        int pos = iMul(y, extremaMap.width) + x;
        int value = extremaMap.data[pos];
        markExtremumAt(value, octave, interval);
        extremaMap.data[pos] = value;
      }
    }
}

void createExtremaMap(CsParams& params, ExtremaMap& extremaMap, BlobMap& blobMap) {
  dim3 blockDim(20, 19);

  for (int octave = 0; octave < params.octaves; ++octave) {
    dim3 gridDim;
    int sampleStep = iMulS(params.initSampleStep, octave);
    int border = 3 * (iMulS(2, octave) * params.intervals + 1) / 2 + 1;
    gridDim.x = iDivUp((extremaMap.width - 2 * border - 2 * sampleStep) / sampleStep, blockDim.x);
    gridDim.y = iDivUp((extremaMap.height - 2 * border - 2 * sampleStep) / sampleStep, blockDim.y);

    findExtrema<<<gridDim, blockDim>>>(params, extremaMap, octave, border, sampleStep);
    SC(cudaThreadSynchronize());
    CK();
  }
}

void freeExtremaMap(ExtremaMap& extremaMap) {
  SC(cudaFree(extremaMap.data));
}

#endif // _EXTREMA_MAP_CUH_
```

ipoint_vector.cuh

```
#ifndef _IPOINT_VECTOR_CUH_
#define _IPOINT_VECTOR_CUH_

#include "common.h"
#include "blob_map.cuh"
#include "extrema_map.cuh"

__global__
void buildIPointVectorPass1(CsParams p, ExtremaMap extremaMap, uint* counters) {
  uint x = iMulS(blockIdx.x, 9) + threadIdx.x;
  uint pos = x;
  uint numIPoints = 0;

  if (x < extremaMap.width) {
    for (uint y = 0; y < extremaMap.height; ++y) {
      int value = extremaMap.data[pos];
      pos += extremaMap.width;

      for (int octave = 0; octave < p.octaves; ++octave)
        if (0 <= hasExtremumAt(value, octave))
          ++numIPoints;
    }

    counters[x] = numIPoints;
  }
}

__global__
void buildIPointVectorPass2(CsParams p, ExtremaMap extremaMap, uint* counters, float4* iPoints) {
  uint x = iMulS(blockIdx.x, 9) + threadIdx.x;
  uint pos = x;

  if (x < extremaMap.width) {
    uint iPointIdx = counters[x];

    for (uint y = 0; y < extremaMap.height; ++y) {
      int value = extremaMap.data[pos];
      pos += extremaMap.width;

      for (int octave = 0; octave < p.octaves; ++octave) {
        int interval = hasExtremumAt(value, octave);
        if (interval >= 0) {
          iPoints[iPointIdx] = make_float4(x, y, interval, octave);
          ++iPointIdx;
        }
      }
    }
  }
}

__global__
void scanArray(uint* data, uint size) {
  extern __shared__ uint block[];
  uint sum = 0;

  for (uint offset = 0; offset < size; offset += blockDim.x) {
    uint pos = offset + threadIdx.x;
    if (pos < size) block[threadIdx.x] = data[pos];

    __syncthreads();

    if (threadIdx.x == 0)
      for (uint i = 0; i < blockDim.x; ++i) {
        sum += block[i];
        block[i] = sum;
      }

    __syncthreads();

    if (pos < size) data[pos] = block[threadIdx.x];
  }
}

__global__
void refineIPoints(CsParams p, float4* iPoints, uint numIPoints) {
  uint idx = iMul(blockIdx.x, blockDim.x) + threadIdx.x;
  float4 iPoint;

  if (idx < numIPoints) {
    Matrix3x3 H, invH;
    float3 d, off;

    iPoint = iPoints[idx];
    uint o = (uint)iPoint.w;
    uint i = (uint)iPoint.z;
    float x = iPoint.x;
    float y = iPoint.y;

    /* Save blob response in the IPoint. */
    iPoint.w = getBlob(o, i, x, y);
    float v = fabsf(iPoint.w);

    uint step = iMulS(p.initSampleStep, o);

    /* Calculate negative 1st order derivates in x, y and scale s. */
    d.x = -0.5f * (getAbsBlob(o, i, x + step, y) - getAbsBlob(o, i, x - step, y));
```

```
    d.y = -0.5f * (getAbsBlob(o, i, x, y + step) - getAbsBlob(o, i, x, y - step));
    d.z = -0.5f * (getAbsBlob(o, i + 1, x, y)    - getAbsBlob(o, i - 1, x, y)   );

    /* Calculate 3D Hessian matrix around the IPoint. */
    H.c11 =  getAbsBlob(o, i, x - step, y) - 2 * v + getAbsBlob(o, i, x + step, y); /* 2nd deriv x, x */
    H.c12 = (getAbsBlob(o, i, x + step, y + step)
           - getAbsBlob(o, i, x - step, y + step)
           - getAbsBlob(o, i, x + step, y - step)
           + getAbsBlob(o, i, x - step, y - step)) * 0.25f;                          /* 2nd deriv x, y */
    H.c13 = (getAbsBlob(o, i + 1, x + step, y)
           - getAbsBlob(o, i + 1, x - step, y)
           - getAbsBlob(o, i - 1, x + step, y)
           + getAbsBlob(o, i - 1, x - step, y)) * 0.25f;                             /* 2nd deriv x, s */
    H.c21 = H.c12;                                                                   /* 2nd deriv y, x */
    H.c22 =  getAbsBlob(o, i, x, y - step) - 2 * v + getAbsBlob(o, i, x, y + step); /* 2nd deriv y, y */
    H.c23 = (getAbsBlob(o, i + 1, x, y + step)
           - getAbsBlob(o, i + 1, x, y - step)
           - getAbsBlob(o, i - 1, x, y + step)
           + getAbsBlob(o, i - 1, x, y - step)) * 0.25f;                             /* 2nd deriv y, s */
    H.c31 = H.c13;                                                                   /* 2nd deriv s, x */
    H.c32 = H.c23;                                                                   /* 2nd deriv s, y */
    H.c33 = getAbsBlob(o, i - 1, x, y) - 2 * v + getAbsBlob(o, i + 1, x, y);        /* 2nd deriv s, s */

    /* Calculate inverse of the Hessian matrix explicitely. */
    /*
       ( 0 1 2 )   ( a b c ) -1               ( e i - f h     c h - b i     b f - c e )
       ( 3 4 5 ) = ( d e f )     = 1/det(A) * ( f g - d i     a i - c g     c d - a f )
       ( 6 7 8 )   ( g h i )                  ( d h - e g     b g - a h     a e - b d )

       det(A) = a e i  +  b f g  +  c d h  -  c e g  -  b d i  -  a f h
    */
    float invDet = 1.0f / (  H.c11*H.c22*H.c33 + H.c12*H.c23*H.c31 + H.c13*H.c21*H.c32
                           - H.c13*H.c22*H.c31 - H.c12*H.c21*H.c33 - H.c11*H.c23*H.c32);
    invH.c11 = invDet * (H.c22*H.c33 - H.c23*H.c32);
    invH.c12 = invDet * (H.c13*H.c32 - H.c12*H.c33);
    invH.c13 = invDet * (H.c12*H.c23 - H.c13*H.c22);
    invH.c21 = invDet * (H.c23*H.c31 - H.c21*H.c33);
    invH.c22 = invDet * (H.c11*H.c33 - H.c13*H.c31);
    invH.c23 = invDet * (H.c13*H.c21 - H.c11*H.c23);
    invH.c31 = invDet * (H.c21*H.c32 - H.c22*H.c31);
    invH.c32 = invDet * (H.c12*H.c31 - H.c11*H.c32);
    invH.c33 = invDet * (H.c11*H.c22 - H.c12*H.c21);

    /* Explicitely calculate interpolation offset with off = invH * d. */
    off.x = invH.c11*d.x + invH.c12*d.y + invH.c13*d.z;
    off.y = invH.c21*d.x + invH.c22*d.y + invH.c23*d.z;
    off.z = invH.c31*d.x + invH.c32*d.y + invH.c33*d.z;

    if (fabs(off.x) < 0.5f  &&  fabs(off.y) < 0.5f  &&  fabs(off.z) < 0.5f) {
      /* Point is sufficiently close to the actual extremum. */
      iPoint.x += off.x * step;
      iPoint.y += off.y * step;
      /* scale = base filter scale / base lobe size * current lobe size */
      iPoint.z = (1.2f/3.0f) * (iMulS(2, o) * (i+off.z+1.0f) + 1.0f);
    } else {
      /* Offset too large. Mark as neglected. */
      iPoint.x = -1.0f;
      iPoint.y = -1.0f;
      iPoint.z = -1.0f;
    }
  }

  __syncthreads();

  /* Write interpolated IPoint position back to IPoint vector. */
  if (idx < numIPoints) iPoints[idx] = iPoint;
}

uint createIPointVector(CsParams& params, ExtremaMap& extremaMap, float4** iPoints) {
  uint* d_counters;
  uint size = extremaMap.width;
  /* Alloc an additional value to shift inclusive scan to an exclusive scan later. */
  SC(cudaMalloc((void**)&d_counters, (size + 1) * sizeof(uint)));
  SC(cudaMemset(d_counters, 0, (size + 1) * sizeof(uint)));
  /* Reserve first uint value and start at second one. */
  ++d_counters;

  dim3 blockDim(512);
  dim3 gridDim(iDivUp(extremaMap.width, blockDim.x));

  buildIPointVectorPass1<<<gridDim, blockDim>>>(params, extremaMap, d_counters);
  SC(cudaThreadSynchronize());
  CK();

  /* Do inclusive scan. */
  scanArray<<<1, blockDim, blockDim.x * sizeof(uint)>>>(d_counters, size);
  SC(cudaThreadSynchronize());
  CK();

  /* Get last value (contains number of detected interest points). */
  uint numIPoints;
  SC(cudaMemcpy(&numIPoints, &d_counters[size - 1], sizeof(uint), cudaMemcpyDeviceToHost));

  if (!numIPoints) return 0;

  /* Allocate memory for ipoint vector. */
  SC(cudaMalloc((void**)iPoints, numIPoints * sizeof(float4)));
  /* Set array start to the reserved uint value to obtain exclusive scan. */
```

```
  --d_counters;

  buildIPointVectorPass2<<<gridDim, blockDim>>>(params, extremaMap, d_counters, *iPoints);
  SC(cudaThreadSynchronize());
  CK();

  return numIPoints;
}

void interpolateIPoints(CsParams& params, float4* iPoints, uint numIPoints) {
  refineIPoints<<<iDivUp(numIPoints, 192), 192>>>(params, iPoints, numIPoints);
  SC(cudaThreadSynchronize());
  CK();
}

#endif // _IPOINT_VECTOR_CUH_
```