Highspeed Videoverarbeitung

Hochgeschwindigkeitskameras werden oft zur Überwachung sehr schneller, vollautomatisierter, industrieller Prozesse eingesetzt. Im Zuge der Marktanforderung, dass die einmal aufgenommenen Prozesse immer schneller zu analysieren sind, soll das aufgenommene Bildmaterial nicht nur zur Dokumentation des Ablaufes dienen, sondern soll vielmehr Bildverarbeitung in Echtzeit verwendet werden. Dabei werden die Bilddaten mit hoher Bildrate (z.B. 200 Bilder pro Sekunde) an einen Rechner übertragen, auf diesem mit neu entwickelten, parallelen, adaptiven Algorithmen analysiert und bei Bedarf Ereignisse in Echtzeit ausgelöst, welche den Automationsprozess steuern. Die Herausforderungen dabei liegen primär in der effizienten Verarbeitung der hohen Datenmengen, welche in Zukunft durch ansteigende Bildraten noch erhöht werden sollen.

Martin Schindler, Christoph Stamm | christoph.stamm@fhnw.ch

In diesem Beitrag beschreiben wir ein gemeinsames Projekt mit dem in Baden-Dättwil ansässigen Unternehmen AOS Technologies. AOS ist ein Anbieter von Hochgeschwindigkeitskameras und dazu passender Software. Die Software-Reihe PROMON von AOS dient der optischen, industriellen Prozessüberwachung [PRO]. Unser Projekt Promon200[1] knüpft an PROMON an, mit dem Hauptziel, die übertragenen und/oder abgespeicherten Videodaten automatisch nach Unregelmässigkeiten in den überwachten Prozessen abzusuchen.

Die belgische Firma Tesin bietet mit ihrer Cyclocam[2] ein mobiles System bestehend aus einer Hochgeschwindigkeitskamera und einem Display an, um zyklische, industrielle Prozesse aufzuzeichnen, direkt vor Ort zu begutachten und eventuell anzupassen. Um ein Hochgeschwindigkeitsvideo mit beispielsweise 1000 Bildern pro Sekunde bei normaler Videobetrachtungsgeschwindigkeit von 25 Bildern pro Sekunde betrachten zu können, werden also 40 Sekunden Betrachtungszeit pro einer Sekunde Aufnahmedauer benötigt. Daher macht es Sinn, uninteressante Teile des Videos automatisch zu überspringen und so die Betrachtungszeit möglichst gering zu halten. Welche Passagen einer Sequenz für den Betrachter von Interesse sind, legt er selber mithilfe von Markierungen fest.

Unser Projekt unterscheidet sich von der Cyclocam dahingehend, dass wir nicht primär uninteressante Teile des Videos überspringen wollen, sondern mehrere Sequenzen eines zyklischen Prozesses vergleichen wollen, um wichtige Unterschiede zwischen den Sequenzen, also Unregelmässigkeiten im Ablauf automatisch feststellen zu können. Zudem wollen wir die Sequenzlänge anhand der sich wiederholenden Bilder möglichst genau selber ermitteln.

Aus verschiedenen Untersuchungen ist bekannt, dass Menschen bei der Betrachtung von zyklischen und langandauernden Videoaufnahmen sehr schnell ermüden und somit an Konzentration einbüssen. Dadurch passiert es schnell, dass die relevanten Unregelmässigkeiten im Ablauf schlicht übersehen werden. Hier kann ein automatisiertes System Abhilfe schaffen, falls es gelingt, dem System klar zu machen, was akzeptable und was unerlaubte Prozessabweichungen sind.

In sehr schnell ablaufenden Prozessen werden hohe Bildraten (> 200 Bilder pro Sekunde) und möglichst kurze Belichtungszeiten benötigt, um alle relevanten Details einzufangen und Bewegungsunschärfen zu vermeiden. Solche hohe Bildraten erhöhen die Schwierigkeit der Aufgabenstellung der automatisierten Prozessüberwachung wesentlich, weil für die durchschnittliche Bearbeitungszeit pro Bild weniger als 5 ms zur Verfügung stehen. Heutige Hochleistungs-PCs sind zwar in der Lage, innerhalb von 5 ms wenige Millionen von Grundoperationen auszuführen. Doch darf man dabei nicht übersehen, dass ein Bild oft aus einer Million oder mehr Bildpunkten besteht. Dies relativiert die Grösse der Anzahl Operationen, die ausgeführt werden können sehr schnell.

Ein automatisiertes, optisches Prozessüberwachungssystem für sehr schnell ablaufende Prozesse muss also in der Lage sein, innerhalb kürzester Zeit korrekte von falschen Prozessabläufen zu unterscheiden. Dies kann beispielsweise auf drei verschiedene Arten von statten gehen:

- In einem ersten Ansatz werden vom Operateur interessante Bildregionen markiert und mit einer Bildverarbeitungsoperation versehen. Eine solche Operation kann beispielsweise die Helligkeit im markierten Bereich bestim-

1 Das Projekt Promon200 ist vom Aargauer Forschungsfonds finanziell unterstützt und begleitet worden.
2 Tesin N.V. Cyclocam: http://www.tesin.be/

men oder die Frequenz einer aufleuchtenden Lampe ermitteln. Die Ausgaben dieser Bildverarbeitungsoperationen können dann mit Schwellwerten verglichen, zu Booleschen Resultaten umgewandelt und schliesslich mit Booleschen Operatoren verknüpft werden. So entsteht am Ende einer Verarbeitungskette ein Signal, welches angibt, ob ein Bild zu einem fehlerhaften Ablauf gehört.

- In einem zweiten Ansatz wird dem System anhand von Videosequenzen gezeigt, wie korrekte und wie falsche Abläufe aussehen können. Diese Sequenzen werden dann dazu verwendet, um den aktuellen Ablauf bildweise mit den gegebenen Sequenzen zu vergleichen. Bei Überschreiten einer gewissen Vergleichstoleranz werden die Bilder markiert und dem Operateur für eine genauere Überprüfung vorgelegt.
- In einem dritten Ansatz versucht das System selbständig die Repetitionen im Ablauf zu ermitteln. Wenn dies gelingt, so können entweder die Einzelbilder eines Ablaufs mit den entsprechenden Einzelbildern des vorgängigen Ablaufs automatisch auf grössere Abweichungen untersucht werden oder es wird über die ganze Sequenz hinweg eine Kennzahl berechnet, welche dann mit den bisherigen Kennzahlen der vorangegangenen Sequenzen verglichen werden kann.

Um einen der drei genannten Ansätze in Echtzeit ausführen zu können, braucht es äusserst effiziente Algorithmen, welche die Parallelität heutiger Computersysteme voll ausnutzen können und darüber hinaus in der Lage sind, die enormen Rechenkapazitäten, welche auf Grafikprozessoren zur Verfügung gestellt werden, für die eigenen Zwecke zu nutzen.

Die Verwendung vorgefertigter Bildverarbeitungsbibliotheken oder -komponenten von bekannten Anbietern wie beispielsweise *MathWorks*[3] oder *National Instruments*[4] wird in diesem Projekt aus dreierlei Gründen ausgeschlossen: Erstens soll PROMON ein eigenständiges Produkt bleiben, welches nicht in wesentlichen Teilen von Fremdkomponenten abhängen darf und zweitens sollen die neusten Entwicklungen im Bereich des *General Purpose Computing on Graphics Processing Units* (GPGPU) ohne lange Entwicklungszyklen in die bestehende Software einfliessen können. Schliesslich drittens, dient das vorliegende Projekt auch zu einem grossen Teil dem Knowhow Transfer im Bereich der parallelen Bildverarbeitung zwischen Hochschule und Industrie.

Sitzt der Schraubverschluss richtig?

Unter dem Begriff der industriellen Prozessüberwachung lässt sich Verschiedenes verstehen. Daher ist es angebracht, an dieser Stelle ein typisches Anwendungsszenario vorzustellen.

In einer Abfüllanlage für Getränkeflaschen soll der Prozess des Verschliessens von Flaschen mit Schraubverschluss überwacht werden. Bei diesem Ablauf kommt es immer wieder vor, dass Deckel beim Aufschrauben kaputt gehen oder dass bereits defekte Deckel auf die Flaschen aufgeschraubt werden. Die Aufgabe einer automatischen Prozessüberwachung besteht also darin, defekte Schraubverschlüsse zu detektieren.

Eine Hochgeschwindigkeitskamera mit passender Streaming- und Überwachungs-Software, wie beispielsweise PROMON, kann also dazu verwendet werden, um die problematischen Teile des Prozesses auf Video aufzunehmen. Da oft nicht im Vornherein klar ist, zu welchem Zeitpunkt es zu einer Fehlfunktion kommen wird, müssen über einen grösseren Zeitraum (z.B. 10 Stunden) Aufnahmen gemacht werden, die dann später, also offline, einer genauen Analyse unterzogen werden. Damit das menschliche Auge die fehlerhaften Abläufe überhaupt wahrnehmen kann, müssen Videoaufnahmen mit einer Bildrate von 200 Bildern pro Sekunde um einen Faktor sechs bis acht verlangsamt werden. Diese manuelle Durchsicht ist ein sehr zeitaufwändiger und ermüdender Vorgang. Daraus ergibt sich die Forderung nach einer automatisierten Lösung, welche die fehlerhaften Stellen möglichst autonom auffinden kann und dabei nicht mehr Zeit benötigt als ein Mensch.

Promon200

Promon200[5] versucht die zuvor genannten Ziele zu erreichen und darüber hinaus, den Übergang von der offline zur online Überwachung zu ermöglichen. Eine online Überwachung setzt nun aber voraus, dass die Bildanalyse mit der Bildrate der Hochgeschwindigkeitskamera Schritt halten kann. Eine zuverlässige online Überwachung erlaubt also nicht nur das viel schnellere Erkennen von Fehlabläufen im Prozess, sondern ermöglicht die Resultate der Echtzeitanalyse auch für die adaptive Anpassung und Optimierung des laufenden Prozesses zu verwenden.

Promon200 ist visuelles Werkzeug (Abb. 1). Es besteht im Wesentlichen aus vier Teilen: auf der linken Seite sind die vorhandenen Bildverarbeitungsalgorithmen, Booleschen Operatoren und Komparatoren aufgelistet; in der Mitte ist der Designer ersichtlich, mit dem die verschiedenen Bildverarbeitungsaufgaben zu einem Gesamtsystem zusammengebaut werden können; auf der rechten Seite oben ist das Livevideobild ersicht-

3 MathWorks, Computer Vision System Toolbox: http://www.mathworks.ch/products/computer-vision/?s_cid=global_nav

4 National Instruments, Vision Builder for Automated Inspection: http://www.ni.com/vision/vbai.htm

5 Promon200 ist die Software, welche im gleichnamigen Projekt innerhalb zwei Jahren entwickelt worden ist.

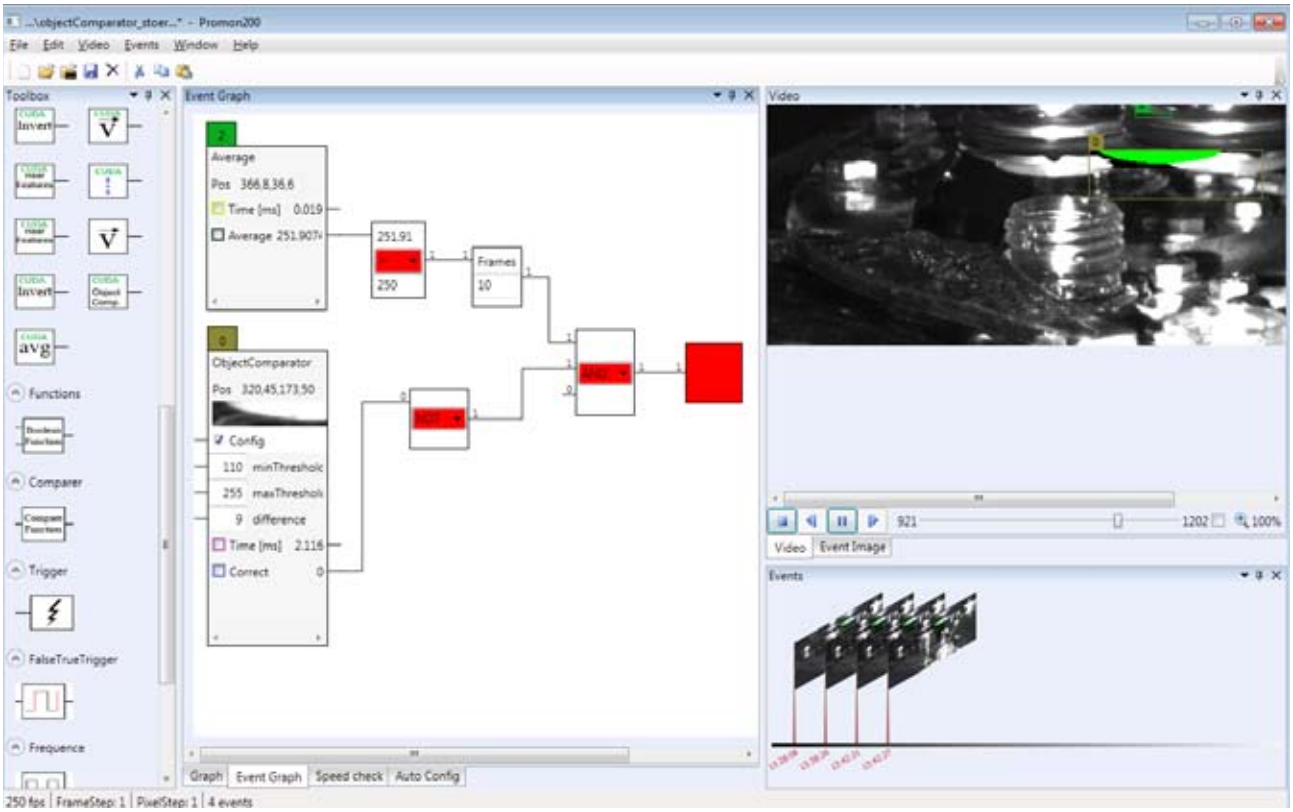

Abbildung 1: Defekter Deckel löst in Promon200 ein Triggersignal aus. Der aktuelle Zustand wird mit allen Messwerten als Ereignis abgespeichert.

lich und darunter sind die ausgelösten Ereignisse in Form von Flaggen aufgelistet.

Die Filterkette innerhalb des Designers zeigt, dass zwei unterschiedliche Regionen (grün und braun markiert) im Video überwacht werden. Sie ist vom Benutzer der Software mittels Drag-and-Drop zusammengestellt worden. Sobald die Ausgänge der beiden Filter einen benutzerdefinierten Schwellwert über- bzw. unterschreiten, wird ein Trigger-Signal ausgelöst, welches ein Ereignis generiert. Diese Ereignisse (Videoframe und der Zustand der Filterkette mit den zum Ereignis abgespeicherten Werten) können entweder in Echtzeit zur Prozesssteuerung dienen oder zu einem späteren Zeitpunkt abgerufen, analysiert und zur Verbesserung des Produktionsablaufs verwendet werden. Da Promon200 über eine Plug-in-Architektur verfügt, können nachträglich neu entwickelte Bildverarbeitungsoperationen (Filter) dem System einfach hinzugefügt werden, ohne dass das gesamte Framework aktualisiert werden muss.

Auf die folgenden drei Aspekte von Promon200 werden wir in diesem Artikel genauer eingehen:

- Detektion von Anomalien in repetitiven, industriellen Prozessen;
- adaptive Echtzeitalgorithmen mit automatischer Parametrisierung;
- Video- bzw. Bildverarbeitung mit Hilfe der GPU.

Erkennung von Anomalien

In der Einleitung haben wir bereits drei grundsätzliche Arten vorgestellt, wie Anomalien in sich wiederholenden, industriellen Prozessen detektiert werden können. Hier wollen wir vor allem auf den dritten Ansatz eingehen, wo das System selbständig die Repetitionen im Ablauf zu ermitteln versucht und dann mithilfe dieser Sequenzlänge in der Lage ist, eine charakteristische Kennzahl für die Sequenz zu ermitteln. Diese Kennzahl lässt sich dann einfach mit denen der vorangegangenen Sequenzen auf grössere Abweichungen untersuchen. Der grosse Vorteil dieses Ansatzes gegenüber dem zweiten Ansatz mit den vorgegebenen korrekten und falschen Sequenzen ist, dass nur die *Region of Interest* gewählt und nicht zuerst eine Datenbasis von falschen und korrekten Sequenzen erstellt werden muss. Zudem schränkt der dritte Ansatz die Arten von Anomalien, welche erkannt werden können, nicht von vornherein ein. Anderseits hängt seine Qualität wesentlich davon ab, ob es gelingt, die Sequenzlänge genau zu ermitteln und ob die Funktion zur Ermittlung der Kennzahl sensitiv genug ist.

Beim ersten Ansatz werden vom Operateur interessante Bildregionen markiert und mit untereinander verknüpften Bildverarbeitungsoperationen versehen. Hierbei geht man davon aus, dass bereits falsche Sequenzen vorliegen oder dass mit der Anomalie verbundene Nebenerscheinungen bekannt sind, welche sich detektieren lassen. Der Operateur leistet hier im Wesentlichen

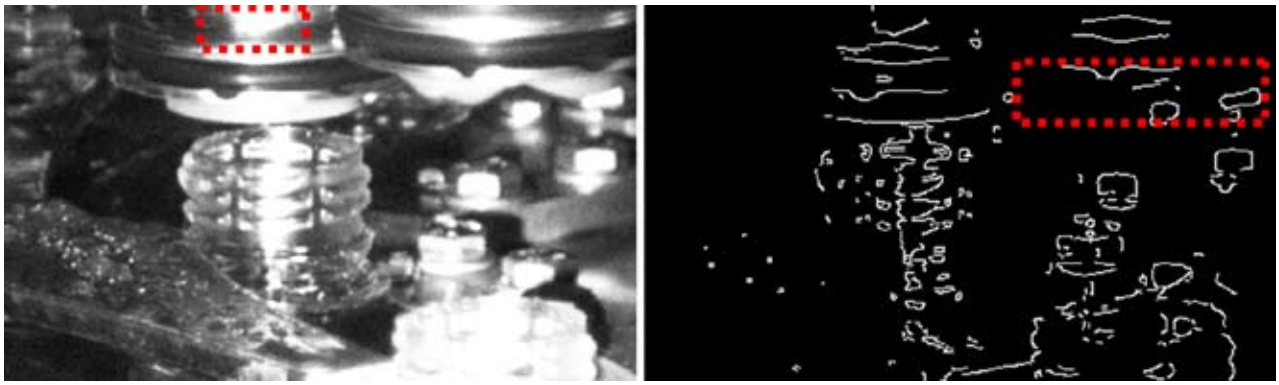

Abbildung 2: links: Graustufenbild mit markierter Region zur Frequenzdetektion; rechts: Kantenbild mit benutzter Region zur Berechnung der HOG-Feature-Vektoren.

eine vorgängige Analysearbeit, um signifikante Merkmale zu finden, anhand derer eine Anomalie erkannt werden kann. Eine solche Analyse ist oft schwierig und zeitaufwendig, kann aber durchaus auch bereits erste Hinweise zur Verbesserung des industriellen Prozesses bringen und somit sehr erwünscht sein. Ein weiterer kleiner Vorteil gegenüber dem selbstlernenden Ansatz ist, dass auf eine Bestimmung der korrekten Sequenzlänge verzichtet werden kann. Der grosse Nachteil hingegen ist wieder wie vorher, dass nur bereits bekannte Problemfälle detektiert werden.

Da der dritte Ansatz ein selbstlernender ist, braucht es einen gewissen Vorlauf, eine erste Lernphase von ein paar Hundert Bildern, um die Wissensbasis aufzubauen, welche anschliessend verwendet wird. Eine solche Lernphase kann in gewissen Situationen störend sein, ist in unserem Ansatz aber nicht weiter störend, da bereits für die Anpassung der Parametrisierung an die vorhandene Hardware eine Kalibrierungsphase vorgeschaltet wird.

Wie bereits erwähnt, müssen wir also im dritten, selbstlernenden Ansatz eine sensitive Funktion für die Sequenz zur Verfügung stellen, welche relevante Veränderungen zwischen den Sequenzen anzeigen kann. Eine solche Funktion basiert oft auf einem oder mehreren *Features*, wie zum Beispiel die durchschnittliche Helligkeit, oder das *Histogram of oriented gradients*[6]. In unserer sensitiven Funktion verwenden wir das HoG, weil wir davon ausgehen, dass sich der Inhalt eines Bildes aufgrund der darin enthaltenen Kantenrichtungen gut charakterisieren lässt.

Die Intensitäten und Richtungen von Kanten können mithilfe der ersten Ableitung über ein Bild einfach berechnet werden (Abbildung 2 rechts zeigt ein Kantenbild). Aus den Kantenrichtungen lässt sich anschliessend für alle Bildpunkte mit relevanter Kantenintensität eines ausgewählten Bereiches ein Histogramm mit acht Richtungsklassen (das HoG) erstellen (Abb. 3). Pro Video bild entsteht dadurch ein charakteristischer Vektor der Länge acht. Alle solchen Vektoren einer Sequenz werden dann in eine konstante Anzahl k Blöcke unterteilt, von denen jeweils ein Minima- und Maximavektor der Länge acht über alle Bilder im Block ermittelt werden. Ein Minima- bzw. Maximavektor gibt also den Minimal- bzw. Maximalwert für jede der acht Richtungen des HoG-Vektors aus den k Bildern des Blocks an. Pro Sequenz werden somit insgesamt $2 \cdot 8 \cdot k$ HoG Werte in einem Sequenz charakterisierenden Vektor zusammengefasst. Dieser Vektor ist der Rückgabewert der charakterisierenden Funktion.

Während der Lernphase wird ein entsprechender Referenzvektor der gleichen Länge $2 \cdot 8 \cdot k$ aus allen Frames der Lernphase gebildet. Dieser Referenzvektor wird mit dem Sequenz charakterisierenden Vektor der ersten Sequenz initialisiert und dann nach jeder Sequenz aktualisiert, wobei bei den Werten aus den Minimavektoren weiterhin das Minimum und bei den Werten der Maximavektoren das Maximum verwendet werden. Nach der Lernphase wird dann für jede Sequenz die charakterisierende Funktion berechnet und mit dem Referenzvektor verglichen. Solange die in der Sequenz bestimmten Min-Max-Wertebereiche vollständig innerhalb der Min-Max-Wertebereiche des Referenzvektors liegen, wird die Sequenz als normal betrachtet, d.h. sie weicht nicht wesentlich von den Sequenzen während der Lernphase ab. Nur wenn alle Sequenzen der Lernphase als korrekt bezeichnet worden sind, darf

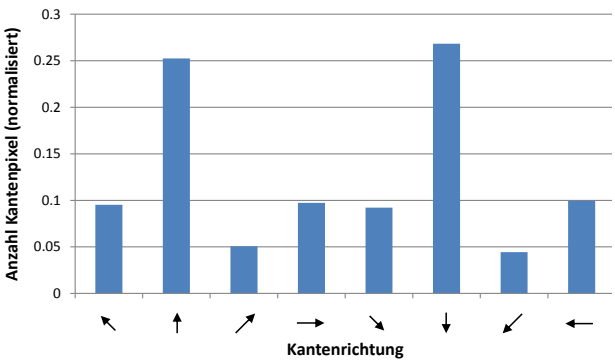

Abbildung 3: Histogramm der Gradienten (HOG)

6 http://en.wikipedia.org/wiki/Histogram_of_oriented_gradients

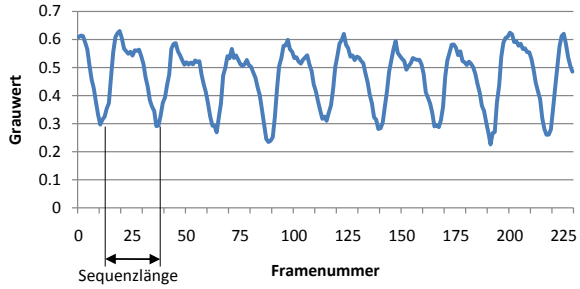

Abbildung 4: Grauwertverlauf über mehrere Frames

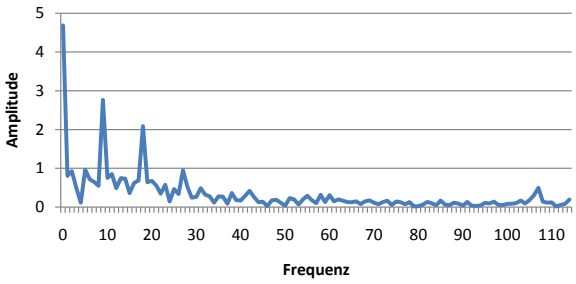

Abbildung 5: Frequenzspektrum des Grauwertverlaufs

angenommen werden, dass die zu überprüfende Sequenz auch als korrekt bezeichnet wird.

Zur Bestimmung der Sequenzlänge oder der Sequenzfrequenz wird während der Lernphase der durchschnittliche Grauwert an einer geeigneten Bildstelle gemessen. Diese Stelle muss sich dadurch auszeichnen, dass die Sequenzfrequenz optisch erkennbar ist (Abb. 4). Aus diesem zeitabhängigen Signal wird dann mittels FFT das Frequenzspektrum (Abb. 5) berechnet und daraus die dominante Frequenz abgelesen. Der erste Peak bei der Frequenz 0 beschreibt die Intensität des Grauwertsignals; der zweite Peak bei der Frequenz 9 zeigt die dominante Frequenz.

Geschwindigkeit und Güte

Wir beschränken uns hier auf ein paar wenige Resultate zum bereits mehrfach erwähnten Beispiel mit den Schraubdeckeln. Das Video besteht aus 42 Sequenzen mit durchschnittlich 25 Bildern der Grösse 640 x 240 Bildpunkten, wobei eine Sequenz das Aufschrauben genau eines Deckels zeigt. Im Video enthalten vier Sequenzen fehlerhafte Deckel, die sich durch ausgefranste Kanten von korrekten Deckeln unterscheiden (Abb. 6).

Für die Berechnung der charakterisierenden Funktion wählen wir eine Filterbereichsgrösse von 143 x 34 Bildpunkten. Damit erreichen wir auf unserem Testsystem eine Verarbeitungsgeschwindigkeit von 540 Bildern pro Sekunde und finden die vier falschen Deckel, ohne weitere Sequenzen als fehlerhaft zu bezeichnen.

Adaptive Parametrierung

Implementierungen von Bildverarbeitungsalgorithmen mit Echtzeitgarantien werden in der Video- und industriellen Bildverarbeitung schon heute eingesetzt. Diese Implementierungen sind jedoch oft plattformspezifisch und erfüllen entweder die Echtzeitbedingung dieser einen Plattform oder eben nicht. Diese duale Sichtweise der Einsatztauglichkeit einer Implementierung ist in der Bildverarbeitung meistens zu hart und kann oft umgangen werden, indem tiefere Bildauflösungen oder kleinere Ausschnitte prozessiert werden, ohne dass sich dabei die Resultate ändern. Das Erstellen von kleineren Bildauflösungen oder Ausschnitten wird daher als Pre-Processing-Schritt betrachtet und kann vom eigentlichen Algorithmus entkoppelt werden (Abb. 7a). Diese Entkopplung ist sehr wertvoll, weil dadurch bestehende, effiziente Bildverarbeitungsalgorithmen unverändert übernommen werden können. Was bei einer solchen Entkopplung jedoch oft verloren geht, ist die formale Spezifizierung der Abhängigkeit von der Laufzeit des Algorithmus. Das heisst, die Wahl der Parameter P1 des Pre-Processings (z.B. die gewünschte Bildauflösung oder die Ausschnittgrösse) ist nicht entkoppelt vom Algorithmus, sondern im Gegenteil, die Laufzeit des gewählten Algorithmus bestimmt indirekt die Parameter P1. Oft werden diese Parameter in meist aufwendigen Performanztests während der Entwicklungsphase ermittelt. Dadurch wird aber die Auswahl der Zielplattformen stark eingeschränkt, weil das Bildverarbeitungssystem seine Tauglichkeit nur auf dem Entwicklungsrechner gezeigt hat.

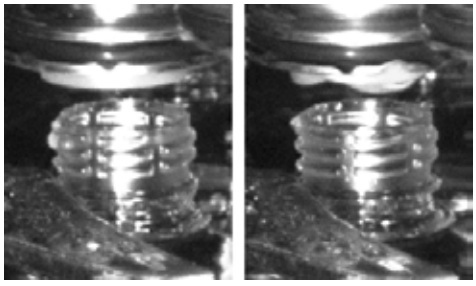

Abbildung 6: Beispiel eines guten (links) und eines fehlerhaften Deckels (rechts)

Anstatt die Parameter P1 manuell zu bestimmen, ist es oft sinnvoller, die passenden Werte dynamisch in einem Rückkopplungsprozess zur Laufzeit zu ermitteln (Abb. 7b). Diese Rückkopplung verschlingt natürlich wiederum Rechenzeit, welche der Hauptaufgabe abgeht. Daher wird in vielen zeitkritischen Anwendungen darauf verzichtet. In gewissen Ausnahmefällen kann jedoch eine dynamische Parametrisierung (z.B. maschinelles Lernen) eingesetzt werden, z.B. wenn eine vorgeschaltete Kalibrierungsphase zu Realbedingungen ausgeführt werden darf. So ist es möglich, die passenden Parameter P1 durch einen eingebauten Lernmechanismus zu ermitteln und für den Echtzeitbetrieb abzuspeichern. Bei Bedarf kann diese Kalibrierungsphase wiederholt werden.

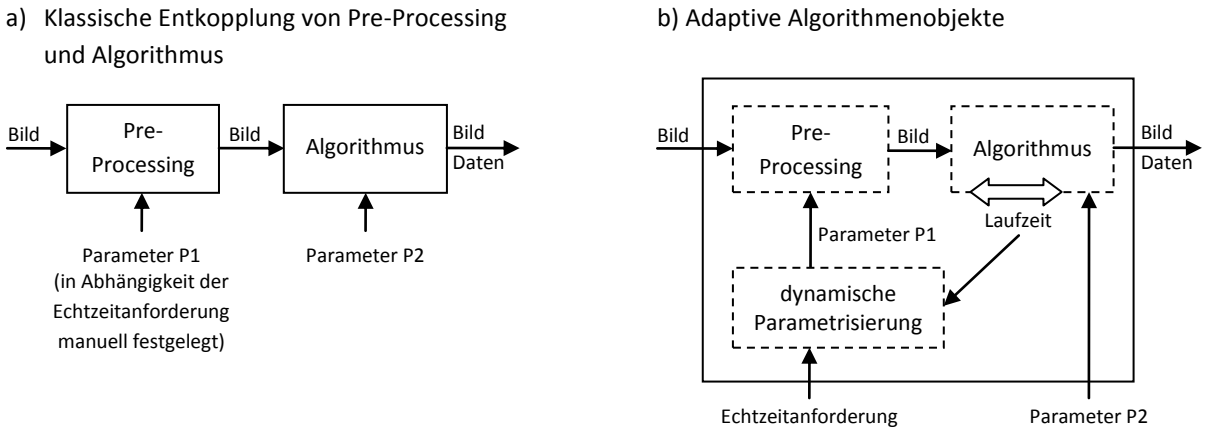

Abbildung 7: Ansätze zur Erfüllung von Echtzeitanforderungen in Algorithmen

Parametrisierung in Promon200

In Promon200 kann die komplette Bildverarbeitungskette an die Echtzeitanforderung, also an eine bestimmte Bildrate und eine maximale Verzögerungszeit, angepasst werden. Solange die gemessene und gemittelte Bildrate grösser oder gleich der gewünschten ist, läuft das System in einem stabilen Zustand. Sobald die gemittelte Bildrate jedoch unter den Sollwert sinkt, müssen geeignete Massnahmen getroffen werden, damit der stabile Zustand wieder hergestellt werden kann.

Welches die geeigneten Massnahmen sind, ist im Normalfall nicht a-priori klar. Daher unterstützt Promon200 drei softwaretechnische (Bildauflösung reduzieren, Videoframes überspringen, Filterregionen verkleinern) und zwei hardwaretechnische Möglichkeiten (Algorithmus auf der GPU ausführen, die Anzahl CPU-Kerne erhöhen). Oberstes Ziel bleibt dabei ständig, fehlerhafte Abläufe bei Aufrechthaltung der geforderten Bildrate zu erkennen. Auf die beiden hardwaretechnischen Massnahmen soll an dieser Stelle nicht eingegangen werden. Im Abschnitt über CUDA zeigen wir dann die spezifische Umsetzung von Bildverarbeitungsalgorithmen für die GPU.

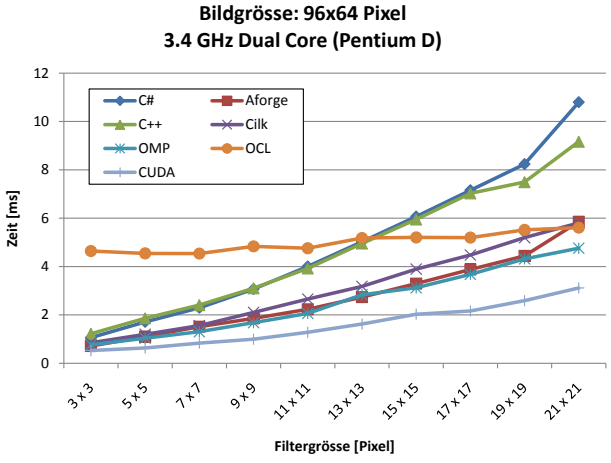

Abbildung 8: Performance eines Boxfilters mit verschiedenen Programmiersprachen und Parallelisierungsbibliotheken.

Dass alle drei softwaretechnischen Massnahmen mit einer offensichtlichen Daten- und Informationsreduktion einhergehen, darf nicht weiter überraschen, wenn man davon ausgeht, dass bei der Entwicklung der Algorithmen das Bestmögliche getan worden ist, um eine gute Effizienz zu erzielen. Entscheidend aber ist, ob die Datenreduktion wirklich zu einer Beeinträchtigung in einem realen Szenario führt, was dann der Fall wäre, wenn infolge der reduzierten Datenmenge nicht mehr alle fehlerhaften Abläufe erkennt werden könnten. Selbstverständlich ist es nicht möglich, alle Beeinträchtigungen infolge der Informationsreduktion im Vornherein auszuschliessen. Solange eine Datenreduktion aber nicht merkbar ist und zu keiner offensichtlichen Beeinträchtigung führt, wird sie vom Benutzer meist stillschweigend akzeptiert (so sind sich Benutzer einer Digitalkamera beispielsweise oft nicht bewusst, dass die Bildspeicherung im JPEG-Format eine Daten- und Informationsreduktion bewirkt).

Anstatt der bedarfsmässigen und automatischen Datenreduktion durch Promon200, könnte

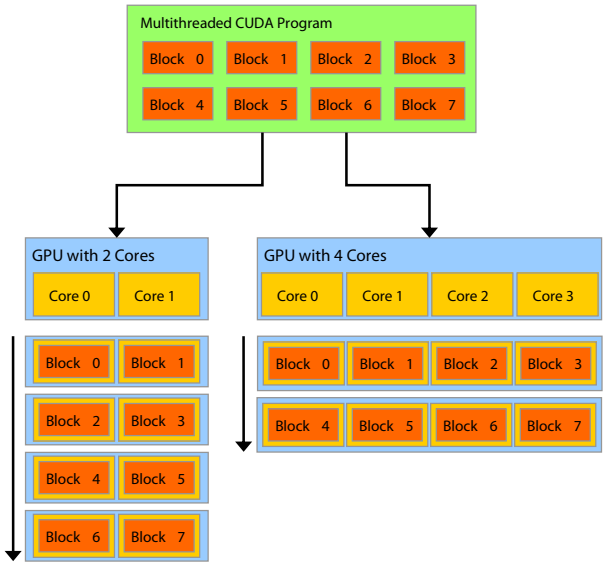

Abbildung 9: Aufteilung eines GPU-Programms in Blöcke (Quelle: [CUDA])

natürlich auch der Benutzer selber die Framerate der Hochgeschwindigkeitskamera reduzieren, eine kleinere Bildauflösung wählen oder den überwachten Bildausschnitt in der Grösse reduzieren. Für den Benutzer erleichtert sich jedoch die Arbeit wesentlich, wenn er alle Einstellungen gemäss seinen Erfahrungen machen kann und anschliessend das System die verschiedenen Datenreduktionsmassnahmen automatisch durchtestet und ihm die Optimierungsvorschläge unterbreitet, welche sich während der Test- oder Kalibrierungsphase von Promon200 bestens bewährt haben.

Bildverarbeitung auf der GPU

Der Grafikprozessor GPU (Graphics Processing Unit) ist primär für die Berechnung der Bildschirmausgabe zuständig. Seit etwa acht Jahren bieten die beiden grössten Grafikkartenhersteller Nvidia und AMD eigene APIs an, mit welchen sich auch andere Aufgaben auf der GPU ausführen lassen. Diese Art der Verwendung von GPUs nennt sich General Purpose Computation on Graphics Processing Units [GPGPU].

Die Leistungsfähigkeit von GPUs in der Bildverarbeitung lässt sich im Vergleich mit der parallelen und der sequentiellen Ausführung eines Algorithmus aufzeigen. In Abbildung 8 zeigen wir die gemittelte Ausführungsdauer eines einfachen Boxfilters mit gegebener Filtergrösse auf einem kleinen Bild der Grösse 96 mal 64 Pixel. Die beiden verwendeten Programmiersprachen (C# und C++) werden sich in sequentieller und in paralleler Ausführung mithilfe von Parallelisierungsbibliotheken (C#: Aforge[7]; C++: Cilk[8], OMP[9]) gegenübergestellt. Im Vergleich dazu zeigt die Grafik auch die Ausführungsdauer auf der GPU (OCL[10], CUDA[11]) inklusive der notwendigen Datentransferzeit. Interessant dabei ist vor allem der hohe Initialaufwand von OpenCL. Unterschiedlichste Performancetests mit verschiedenen GPU-APIs und verschiedenen Programmiersprachen haben gezeigt, dass die Verwendung von OpenCL unabhängig der Filtergrösse einen Initialaufwand von ca. 5 ms mit sich bringt, und somit die maximale Verarbeitungsrate auf der GPU auf ca. 200 Bilder pro Sekunde begrenzt. Infolge dieser Beschränkung verwenden wir die GPU mit einem herstellerspezifischen API, in unserem Fall dem CUDA API von Nvidia.

Ein Grund für den zusätzlichen Performanzgewinn infolge des GPU-Einsatzes liegt in der massiv parallelen Ausführung ohne Synchronisation, die gerade in der Bildverarbeitung anzutreffen ist. Während ein aktueller Prozessor von Intel (i7-980X) sechs Prozessorkerne hat, befinden sich auf dem aktuellen Topmodell von Nvidia (GeForce GTX 590) 1024 Kerne. Eine GTX 480, welche wir für die meisten Performancetests verwendet haben, hat 480 Kerne, eine Rechenleistung von 1345 GFLOPS und eine Speicherbandbreite von 177.4 GByte/s. Im Vergleich dazu hat ein Intel i7-980 vier Kerne, eine Rechenleistung von 80 GFLOPS und eine Speicherbandbreite von 4,8 GByte/s. Diese hohe Speicherbandbreite bei GPUs nützt in den meisten Fällen jedoch wenig, da die Videodaten zuerst vom CPU-RAM zur GPU transferiert werden müssen.

Programmieren mit CUDA

Die grosse Anzahl an Rechenkernen beeinflusst auch wesentlich die Programmierung der Algorithmen. Sie müssen einerseits hoch parallelisierbar sein und anderseits möglichst auf Synchronisation verzichten, damit die Rechenleistung einer Grafikkarte voll ausgenutzt werden kann. Da die GPU der *Single Program Multiple Data* Architektur (SPMD) folgt, kann lediglich ein einziges Programm auf der GPU in Ausführung sein. Das heisst, alle Prozessorkerne führen zwar das gleiche Programm aus, aber mithilfe der Thread-ID können dennoch individuelle Aufgaben pro Thread ausgeführt werden.

In der CUDA-Terminologie wird die in C geschriebene Thread-Funktion eines GPU-Programms als *Kernel*[12] bezeichnet. Die erforderliche Anzahl Ausführungen des Kernels wird beim Start als Parameter in Form von *BlockSize* und *GridSize* übergeben. Die beiden Parameter können ein-, zwei- oder dreidimensional gewählt werden, und bestimmen die Anzahl Kernelaufrufe pro Block und die Anzahl Blöcke. Die Blockgrösse ist bei aktuellen Grafikkarten auf 1024 begrenzt. Der Grund für diese Aufteilung in Blöcke liegt in der Skalierbarkeit. Da Blöcke unabhängig voneinander ausgeführt werden, wird eine GPU mit mehr Kernen das Programm automatisch in kürzerer Zeit ausführen als eine GPU mit weniger Kernen, ohne dass der Code angepasst werden muss (Abb. 9).

7 AForge.NET ist ein Open-Source-Framework für die Bereiche Maschinelles Sehen und Künstliche Intelligenz. Es ist in der .NET-Sprache C# geschrieben. http://www.aforgenet.com/

8 Intel® Cilk™ Plus ist eine Spracherweiterung zu C/C++, welche auf schnelle, einfache und zuverlässige Art erlaubt, die Mehrkernprozesse parallel zu benutzen. http://software.intel.com/en-us/articles/intel-cilk-plus/

9 OpenMP ist eine seit 1997 gemeinschaftlich von verschiedenen Hardware- und Compiler-Herstellern entwickelte Programmierschnittstelle. Der Standard dient zur Shared-Memory-Programmierung in C/C++/Fortran auf Multiprozessor-Computern. http://openmp.org/wp/

10 OpenCL ist eine Schnittstelle für uneinheitliche Parallelrechner, die z.B. mit Haupt-, Grafik- und/oder digitale Signalprozessoren ausgestattet sind, und mit der zugehörigen Programmiersprache „OpenCL C". http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf

11 Cuda API: http://www.nvidia.com/object/cuda_home_new.html

12 Wir verwenden hier den englischen Begriff Kernel in Abgrenzung zum deutschen Wort Kern, welches wir im Zusammenhang mit Prozessorkernen verwenden.

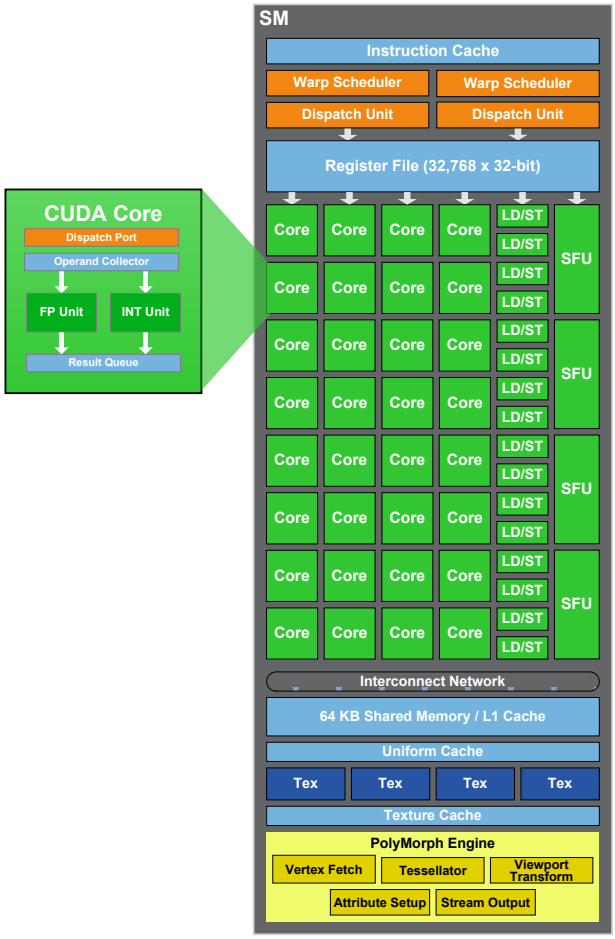

Abbildung 10: Aufbau eines Streaming Multiprocessors mit 32 Kernen. Eine GTX 480 hat 15 dieser SMs (Quelle: [GF100])

Innerhalb eines Blocks haben die Threads einen gemeinsamen, relativ kleinen (mehrere KByte), dafür aber schnellen Speicher (*shared memory*) zur Verfügung. Alle Threads haben zudem Zugriff auf einen gemeinsamen Speicher (*global memory*), der bei aktuellen Grafikkarten mehrere 100 MByte gross ist. Threads werden den sogenannten *Streaming Multiprocessors* (SM) zugewiesenund zwar immer in der Granularität eines Blocks. Die einzelnen Blöcke dürfen keine Abhängigkeiten untereinander haben, da deren Ausführungsreihenfolge nicht garantiert ist. Hingegen können Threads innerhalb eines Blocks an einer Barriere synchronisiert werden. Eine GTX 480 basiert auf der Fermi-Architektur [GF100] und hat 15 SMs mit je 32 Kernen (Abb. 10). Jedem SM können 1536 Threads zugewiesen werden. Das heisst eine optimale Aufteilung wäre zum Beispiel pro SM sechs Blöcke zu je 256 Threads.

Die Wahl der richtigen Blockgrösse kann die Gesamtperformanz eines GPU-Programms wesentlich beeinflussen. So ist aus der Tabelle 1 klar ersichtlich, dass sich die Berechnungsdauer des Skalarprodukts zweier Vektoren der Länge 2^{22} bis zu einem Faktor zehn unterscheiden kann, je nach gewählter Blockgrösse. Der Performancebericht von Nsight[13] in Abbildung 11 zeigt die Auswirkungen der Blockgrösse auf die Auslastung der GPU. Im linken Teil der Abbildung ist ersichtlich, dass bei einer Blockgrösse von 192 eine optimale Auslastung der SMs erreicht werden kann. Im rechten Teil der Abbildung ist die Auslastung der SMs in Abhängigkeit der Blockgrössen grafisch dargestellt. Zusätzlich zu diesen Angaben liefert Nsight genaue Zeitmessungen zu sämtlichen CUDA Funktionsaufrufen.

Der relevante Teil des Programms zur Berechnung des Skalarprodukts ist in Listing 1 ersichtlich. Es zeigt den Kernel für die GPU und die wichtigsten Teile des aufrufenden Hauptprogramms. Das Schlüsselwort *__global__* macht den GPU-Kernel für das Hauptprogramm sichtbar. In diesem feingranulierten Beispiel berechnet jeder Thread des Blocks genau eine Multiplikation zweier Vektorelemente und speichert das Resultat im gemeinsamen Speicherbereich *temp*. Mit dem Aufruf der Prozedur *__syncthreads()* wird sichergestellt, dass alle Threads eines Blocks an der Stelle ihre Multiplikation abgeschlossen haben, bevor der Thread mit der ID 0 die Summe in der lokalen Variable *sumPerBlock* bildet und diesen Wert mit der atomaren Funktion *atomicAdd()* zur globalen Variable *c* addiert. Diese atomare Addition ist notwendig, da mehrere Blöcke gleichzeitig ausgeführt werden können.

Blocksize	Allocation [µs]	Memcopy [µs]	Kernel [µs]	Total [µs]
1024	1496	3509	20126	25131
768	1507	3510	10690	15707
512	1482	3489	2606	7577
256	1496	3511	2052	7059
192	1485	3459	2037	6981
128	1483	3490	3009	7982

Tabelle 1: Zeitmessung am Beispiel des Skalarprodukts mit einer Vektorgrösse von 2^{22}

CUDA Tipps

Wie bereits erwähnt, sollten die typischen Eigenheiten einer GPU, nämlich die grosse Anzahl Rechenkerne, die relativ hohe Ausführungsgeschwindigkeit und der nicht unerhebliche Zeitaufwand beim Datentransfer zwischen Hauptspeicher und Grafikadapter, die Parallelisierung eines Algorithmus beeinflussen. Die hohe Anzahl Kerne spricht für eine recht feine Granularität, sofern diese nicht einen übermässigen Datenaustausch über die Thread-Grenzen hinweg bedingt. Vor allem die Wahl der Blöcke und deren Grösse haben einen grossen Einfluss auf die parallele Ausführungsgeschwindigkeit, weil der Datenaustausch innerhalb eines Blockes wesentlich

13 Nvidia Nsight: http://developer.nvidia.com/nvidia-parallel-nsight

```
#define TPB 512      // threads per block
#define N   TPB*8192 // number of vector elements

__global__ void dotproduct(float* a, float* b, float* c){
      __shared__ float temp[TPB];
      int index = threadIdx.x + blockIdx.x*blockDim.x;
      temp[threadIdx.x] = a[index]*b[index];
      __syncthreads();
      if (0 == threadIdx.x) {
            float sumPerBlock = 0;
            for(int i=0; i<TPB; ++i)
                  sumPerBlock += temp[i];
            atomicAdd(c, sumPerBlock);
      }
}

int main(void){
      // memory allocation and initialization
      ...

      // copy inputvectors to GPU
      cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
      cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

      // kernel launch (<<< Gridsize, Blocksize>>>)
      dotproduct<<< N/TPB, TPB>>>(d_a, d_b, d_c);

      // copy result back from GPU
      cudaMemcpy(h_c, d_c, sizeof(float), cudaMemcpyDeviceToHost);
}
```

Listing 1: Skalarprodukt-Kernel und Teil des aufrufenden Hauptprogramms

schneller ist, als der Zugriff auf den globalen GPU-Speicher. Somit ist bei einer Dekomposition einer Problemstellung in Blöcke darauf zu achten, dass die Dekomposition entlang der natürlichen Datengrenzen erfolgt, z.B. entlang der Eingabe- oder Ausgabedaten. Generell ist darauf zu achten, dass der Speichertransfer zwischen der CPU und der GPU möglichst gering ist, da dieser ca. 10- bis 20-mal langsamer ist als der Zugriff auf das *global memory* der GPU. Wo immer möglich sollte jedoch das *shared memory* anstatt dem *global memory* verwendet werden, um von einer besseren Zugriffszeit profitieren zu können.

Unterschiedliche GPUs unterscheiden sich nicht nur in der Anzahl der Rechenkerne, der Grösse des Speichers und der Taktfrequenz, sondern haben auch unterschiedliche Befehlssätze. Diese Geräteeigenschaften lassen sich über das CUDA-API abfragen. Ebenso ist die Aufteilung der Threads in Blöcke unterschiedlich zu handhaben. Damit die Streaming Multiprocessors während der ganzen Ausführungszeit des GPU-Programms möglichst voll ausgelastet sind, müssen die Blöcke so dimensioniert werden, dass erstens pro SM mehrere Blöcke vorhanden sind, um bei Synchronisation und Speicherzugriff zwischen laufbereiten und wartenden Blöcken schnell wechseln zu können, dass zweitens die Anzahl Threads pro SM ein ganzzahliges Vielfaches der Blockgrösse ist und drittens, dass alle aktiven Blöcke pro SM nicht mehr shared memory und Register benötigen, als ein SM zur Verfügung stellt. Darüber hinaus gibt es noch weitere Bedingungen (z.B. ein Vielfaches der Warp-Size), die möglichst eingehalten werden sollen. Mithilfe des CUDA Occupancy Calculators [OCALC] lassen sich auf einfache Art plattformspezifisch gute Werte im Vorfeld ermitteln.

Nvidia bietet zu ihren CUDA-fähigen Grafikadaptern auch verschiedene Werkzeuge, z.B. ein Debugger und ein Performance-Analyzer. Für die Entwicklungsumgebung Visual Studio von Microsoft gibt es zudem ein Plug-In. Damit lässt sich

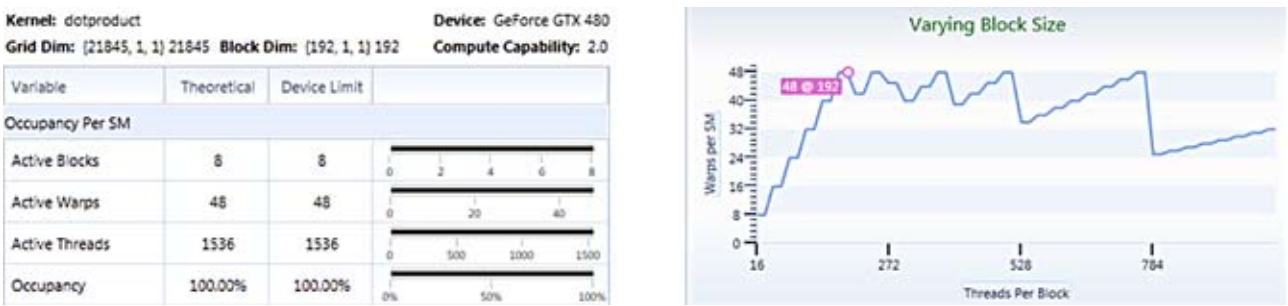

Abbildung 11: Auszug aus dem Performance Report von Nsight

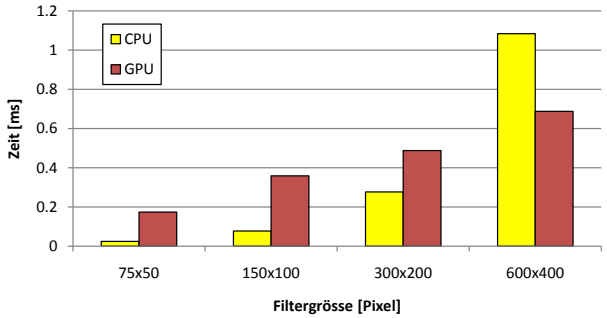

Abbildung 12: Geschwindigkeitsvergleich CPU/GPU des Durchschnittsgrauwertfilters

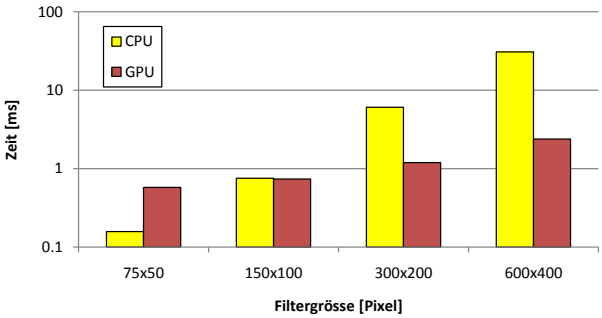

Abbildung 13: Geschwindigkeitsvergleich CPU/GPU des Templatematching-Filters

der Kernelcode debuggen, das heisst die Ausführung des Kernelcodes lässt sich anhalten und ein Wechsel zu einem beliebigen anderen Thread in einem anderen Block ist möglich. Voraussetzung dafür ist allerdings, dass neben einer CUDA-fähigen Grafikkarte noch eine zweite Grafikkarte zur Ansteuerung des Bildschirms vorhanden ist.

Promon200 und CUDA

Mehrere der in Promon200 verfügbaren Filter sind sowohl in einer parallelen C#- als auch in einer CUDA-Version vorhanden. Bei sehr einfachen Filtern mit linearem Aufwand, wie zum Beispiel der Bestimmung der durchschnittlichen Helligkeit, lohnt sich der Einsatz der CUDA-Variante auf unserem Testsystem erst ab einer Filterbereichsgrösse von 400 x 300 Bildpunkten (Abb. 12). Dies liegt daran, dass der Overhead, verursacht durch den zusätzlichen Datentransfer zwischen Hauptspeicher und Video-RAM, den Performanzgewinn bei kleineren Filterbereichen wieder zunichte macht. Bei einem rechenintensiveren Filter, wie beispielsweise dem Templatematching-Filter, lohnt sich die Ausführung auf der Grafikkarte bereits ab einer Filterbereichsgrösse von 150 x 100 Bildpunkten und ab einer Grösse von 300 x 200 Bildpunkten steht dann nur noch die CUDA-Variante zur Verfügung, weil die C#-Variante die Echtzeitforderung von 200 Bildern pro Sekunde nicht mehr einhalten kann (Abb. 13). An diesem Beispiel ist gut ersichtlich, dass in Abhängigkeit des Filtertyps und der Filterbereichsgrösse entschieden bzw. ausgetestet werden muss, welche von den zur Verfügung stehenden Varianten leistungsfähiger ist und somit zum Einsatz kommen soll.

Zusammenfassung und Ausblick

In umfangreichen Performancetests verschiedenster Filterimplementierungen auf der CPU und GPU haben wir die Leistungsfähigkeit unserer Implementierungen nachgewiesen. Für mehrere konkrete Anwendungsbeispiele hat Promon200 genau die fehlerhaften Sequenzen in den Abläufen gefunden. Die dabei geforderte Ablaufgeschwindigkeit von 200 Bildern pro Sekunde kann eingehalten und teilweise stark übertroffen werden. So lässt sich auf einem PC mit i7-Prozessor (3 GHz) und einer Nvidia GTX480 Grafikkarte der exemplarische Produktionsablauf einer Flaschenabfüllanlage mit einer Bildrate von bis zu 540 Frames pro Sekunde überwachen. In diesem Prozess werden Flaschen aussortiert, bei denen der Schraubdeckel fehlerhaft oder nicht korrekt verschraubt ist. Diese Aufgabe ist mit einem vollautomatischen Lern- und Testsystem umgesetzt worden, welches beliebige Anomalien in repetitiven Produktionsabläufen erkennt.

Referenzen

[CUDA] Nvidia CUDA C Programming Guide: http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf

[GF100] Nvidia GF100 White Paper: http://www.nvidia.co.uk/object/IO_89569.html

[GPGPU] General-Purpose Computation on Graphics Hardware: http://gpgpu.org/about

[OCALC] http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

[PRO] PROMON Streaming, AOS Technologies AG: http://www.aostechnologies.com/process-monitoring/products-process-monitoring/promon-streaming/