Praxis der Multikern-Programmierung

Stereo Vision

Team:
Lukas Frank
Maximilian Noppel
Tobias Viehmann

Mail:
uidyk@student.kit.edu
udqlj@student.kit.edu
uwwrp@student.kit.edu

4. Februar 2018

Inhaltsverzeichnis

1 Phase 1: Implementierung Filter **1**
1.1 Ansätze 1
1.2 Gaussfilter 2
1.2.1 Optimierungen 2
1.2.2 Auswertung 3
1.3 Bilateralfilter 3
1.3.1 Optimierungen 3
1.3.2 Auswertung 4
1.4 Allgemeine Schwierigkeiten 4
1.4.1 Probleme Gauss-Filter 5
1.4.2 Probleme Bilateral-Filter 5

2 Phase 2: Implementierung Ähnlichkeitsmaße **6**
2.1 Ansätze 6
2.2 Sum of Squared Differences (SSD) 6
2.2.1 Optimierungen 6
2.2.2 Weitere Optimierungsideen 8
2.3 ZNCC 8
2.3.1 Optimierungen 8
2.3.2 Auswertung 10
2.4 Allgemeine Schwierigkeiten 10
2.4.1 Cuda 10
2.4.2 Probleme ZNCC 10

Abbildungsverzeichnis

1 Visualisierung des SSD Algorithmus 12
2 Performanz des Gaussfilters (BM_G16) 13
3 Performanz des Bilateralenfilters (BM_B6_300) 13
4 Performanz des SSD Algorithmus 14
5 Performanz unterschiedlicher Blockgrößen (ssd_cuda_v5) 15
6 Performanz des ZNCC Algorithmus 16

Tabellenverzeichnis

1 Performanzdaten *BM_G16* des Gaussfilters 11
2 Performanzdaten *BM_BIG_B6_300* des Bilateralfilters 11
3 Performanzdaten des SSD Algorithmus 11
4 Performanzdaten unterschiedlicher Blockgrößen des ssd_cuda_v5 auf *BM_SSD20real_time* 14
5 Performanzdaten des ZNCC Algorithmus 15

Einleitung

Im WS17/18 wurden im Rahmen der Vorlesung "Praxis der Multikern-Programmierung" mehrere Teile der Stereo-Vision Pipline implementiert. Zu zwei gegebenen Bildern, die aus unterschiedlicher Perspektive aufgenommen wurde, galt es die Tiefeninformation der Objekte auf dem Bild zu berechnen. Aus den 5 Stufen der Pipeline *(load images, rectification, preprocess, stereo matching, visualize)* wurde die Vorverarbeitung sowie das eigentliche Stereo-Matching implementiert.

1 Phase 1: Implementierung Filter

Während der ersten Phase wurde die Vorverarbeitung implementiert. Da das Stereo-Matching nur die Struktur des Bildes benötigt, wird das Bild mit einem Hochpass-Filter gefaltet. Zur Vorverarbeitung wurde der **Difference of Gaussian** sowie der **Difference of Bilateral** implementiert.

1.1 Ansätze

Eigene Testumgebung Zur automatisierten Verifizierung unserer Implementierungen war die Überlegung, eine Testumgebung aufzusetzen. Das von unserer Library berechnete Bild wird, mithilfe eines Python-Skripts, getestet. Um das Soll-Bild zu erzeugen, verwenden wir die Bibliothek OpenCV. Das Soll-Bild wird mit unserem Ergebnis anschließend verglichen.

Betriebssysteme Im Laufe des Projektes haben wir unter Windows, Linux und MacOS entwickelt. Deshalb war es nicht möglich, allein mit dem zur Verfügung gestellten Makefile, die Bibliothek auf jedem unserer Systeme zu bauen. Aus diesem Grund haben wir **CMake** in unser Projekt integriert. Nach einiger Einarbeit ermöglichte uns dieses plattformübergreifende Buildsystem in der jeweils bevorzugten Umgebung zu entwickeln. Ein entscheidener Grund für die Auwahl von CMake war die Integration von Cuda. Alle benötigten Informationen befinden sich in der Datei `CMakeLists.txt`.

Algorithmus In beiden Teilen der Projektphase bestand einer der ersten Schritte daraus, sich grundlegend mit den vorliegenden Algorithmen auseinanderzusetzen.

Neben dem Verständnis über den Ablauf der Filter, haben wir dabei gemeinsam die Informationen zusammengetragen, die wir in der Implementierung umsetzten.

Technologieauswahl Wir haben uns überlegt mehrere der vorgestellten Technologien wie SIMD, OpenMP und Cuda zu verwenden. Nach einigen simplen Implementierungen in OpenMP und Cuda haben wir jedoch das größte Potenzial in den Cuda-Implementierungen gesehen. Die Performance sowie die Möglichkeiten zur Optimierung haben dazu geführt, dass wir uns hauptsächlich auf Cuda konzentriert haben.

1.2 Gaussfilter

Unsere erste naive Implementierung des Gauss-Filters bestand darin, den Filter mit einem vorberechneten zweidimensionalen Kernel zu berechnen. Unsere sequentielle Implementierung berechnet mit Hilfe der Gaussfunktion die benötigten Kernelwerte, normalisiert diese und wendet den Kernel auf das gesamte Bild an. Eine simple, jedoch ineffiziente OpenMP sowie eine erste Cuda-Variante ließ sich daraus leicht ableiten.

1.2.1 Optimierungen

Separierbarkeit Gauss-Filter Die erste Optimierung, die wir vorgenommen haben, umfasst die Eigenschaft der Separierbarkeit. Wird der Gauss-Filter anstatt mit einem 2D-Filter durch zwei 1D Filter realisiert, kann die Rechenzeit reduziert werden, wobei die Genauigkeit des Ergebnisses nicht beeinflusst wird. [1]

Fourier-Transformation Wir hatten die Idee, anstatt die Bilder mit einem Filter im Ortsbereich zu falten, den Gauss-Filter im Frequenzbereich durchzuführen. Um dies zu realisieren, muss das Bild mit der *Fourier-Transformation* in den Frequenzbereich überführt werden. Anschließend wird es mit der Transferfunktion multipliziert und das Ergebnis mit der inversen Fourier-Transformation wieder in den Ortsbereich überführt. Wir kamen zu dieser Überlegung mit der Hoffnung, eine bessere Laufzeit erreichen zu können. Wird der Gauss-Filter nicht separiert angewandt, hat man für ein quadratisches Bild mit der Größe n und einem Kernel der Größe m eine Laufzeit von O($n^2 * m^2$). Durch eine Fast-Fourier-Transformation lässt sich die Fourier-Transformation 1D in O($n * logn$) approximieren. Insgesamt für eine

[1] https://de.wikipedia.org/wiki/Separierbarkeit

Berechnung des Gauss-Filters ergibt sich eine Laufzeit von O($4 * n * logn$) (2* (1D-Transformation) * 2 Richtungen) was in der Theorie deutlich schneller wäre als eine Berechnung im Ortsbereich. Diesen Ansatz haben wir relativ spät verworfen, da wir für die Implementierung nicht ausreichend Zeit zur Verfügung hatten und außerdem die Lösung mit CUDA bereits nur wenige Millisekunden dauerte.

1.2.2 Auswertung

Die Ergebnisse der verschiedenen Optimierungen sind in Tabelle 1 erkennbar. Wir konnten eine Verbesserung um den Faktor 1000 erzielen.

1.3 Bilateralfilter

Der naive rechenintensive Bilateralfilter hat deutlich mehr Möglichkeiten zur Optimierung geboten als der Gauss-Filter. Vergleichbar mit der Herangehensweise beim Gaussfilter war eine korrekte sequenzielle Lösung das erste Ziel. Diese erste Implementierung bildete die gesamte Projektphase über die Referenzimplementierung des Test-Skripts.

Analog zu der Gauss-Implementierung wurde dafür ein zweidimensionaler Kernel vorberechnet. Zusätzlich wurde eine Lookup-Table für alle möglichen Farbdistanzen erstellt. Ohne nennenswerte Änderungen des Codes führte die Portierung auf Cuda bereits zu einer Beschleunigung um den Faktor 5.

1.3.1 Optimierungen

Kernelberechnung Eine Vorberechnung der beiden Kernel und die Abspeicherung im *Shared Memory* führt dazu, dass diese Daten nicht von jedem Thread neu berechnet werden müssen. Eine Analyse durch den Profiler hat ergeben, dass es bezüglich der Rechenzeit dabei keinen entscheidenen Unterschied macht, ob die Kernel auf der CPU oder GPU vorberechnet werden.

Weiterhin war ein Ansatz den Gauss-Kernel zu komprimieren, um weniger lesende Zugriffe zu benötigen, die in keinem Fall zu einem *Bank Conflict* führen. Aus dem symmetrischen Aufbau resultiert, dass nur ungefähr ein Achtel des gesamten Kernels nötig ist, um diesen zu rekonstruieren. Nacheinander werden dabei zunächst horizontale, vertikale, diagonale und übrige Pixel betrachtet. Obwohl dies zu einem

geringen Speedup führte, wurde es in den folgenden Versionen des Filters wieder entfernt. Es führte zu schwer lesbarem Code und wurde durch die folgende Optimierung überflüssig. (Tag: bilateral_v4)

Separierbarkeit Mathematisch ist der bilaterale Filter nicht separierbar, in der Praxis aber entstehen resultierende Bilder, die nur minimal von der korrekten Lösung abweichen. Der Rechenaufwand unseres Filters reduzierte sich durch diese Approximation auf ein Achtel der vorherigen Rechenzeit. Desweiteren reduzierte sich die Komplexität. In 2 Kernelaufrufen wird von jedem Block jeweils nur eine Zeile oder Spalte des Bildes betrachtet. Für eine vollständige Auslastung der Multiprozessoren schreiben 1024 Threads pro Block die benötigten Daten zunächst in den *Shared Memory* und berechnen daraufhin vollständig parallel einen Teil des separierten Filters.

Cachelines Der Profiler hat ergeben, dass der Zugriff auf den globalen Speicher im horizontalen Filter sehr gut, im Vertikalen jedoch äußerst inperformant ist. Aus diesem Grund schreibt sowohl der horizontale als auch der vertikale Filter das Ergebnis transponiert.

Pixelgröße Die letzte angewendete Optimierung umfasst die Speicherung der Pixel im *Shared Memory*. Jeder RGB-Pixel des Eingabebildes benötigt 24 Bit zur Speicherung. Da der *Shared Memory* in 4 Byte große Blöcke eingeteilt wird, hat sich gezeigt, dass eine Vergrößerung jedes Pixels auf 4 Byte die Zugriffe verbessert. Das führte zu einem Anstieg der Performance um weitere 20%.

1.3.2 Auswertung

Vorallem die Separierung des Bilateralfilters hat in der Praxis große Vorteile gebracht. Wir konnten hier Abweichungen von nur 3 - 4% erreichen, was für unsere Testfälle ausgereicht hat. Die Performanzdaten sind in der Tabelle 2 ersichtlich.

1.4 Allgemeine Schwierigkeiten

Betriebssysteme Der Einsatz von unterschiedlichen Betriebssystemen zur Entwicklung des Projektes hatte einige Nachteile. Es war eine umfangreiche Einarbeitung in CMake erforderlich, um das Projekt auf Systemen mit und ohne Cuda lauffähig zu bekommen. Insbesondere Windows hat dabei enorme Probleme bereitet. Beispielsweise wird zum Exportieren von Bibliotheks-Methoden das Schlüsselwort

`__declspec(dllexport)` benötigt. Eine Alternative ohne den Code zu verändern, bieten Moduldefinitionsdateien. Ab Version 3.4.3 unterstützt CMake zwar eine automatisierte Generierung, dies funktioniert jedoch nicht mit Cuda-Quelldateien. Eine doppelte Ausführung des CMake-Build-Prozesses ist notwendig geworden.

CUDA Leider hatte ein Großteil des Team keine Nvidia Grafikkarte zur Verfügung und musste daher auf die ATIS-Poolrechner zurückgreifen. Ein Remoterechner wäre dafür sehr von Vorteil gewesen und hätte auch die Arbeit von zu Hause ermöglicht.

1.4.1 Probleme Gauss-Filter

Separierbarkeit Den Filter separiert zu implementieren hat bei uns einige Zeit in Anspruch genommen, da wir lange Zeit einige Fehler nicht finden konnten. Anfangs haben wir die 2D Gaußfunktion verwendet um die Matrix K' als Kernel zu berechnen. Ausgehend von der Matrix war der Ansatz die 2 separierten Kernel K und K^T zu erzeugen. Dafür haben wir angenommen, dass $K * K^T = K'$ gilt, was wegen der Matrixmultiplikation jedoch nicht korrekt ist. Nach Recherche haben wir den Fehler beheben können, indem wir direkt den separierten Kernel mit der 1D Gaussfunktion berechnen.

Negativer Schleifenzähler Eine Schleife im Gaussfilter sollte vom Negativen bis zum Positiven über die Kernelgröße iterieren. Als Zählervariable haben wir allerdings den Datentyp *size_t* gewählt, wodurch die Schleife lediglich im Positiven zählte.

1.4.2 Probleme Bilateral-Filter

Verwendung Shared Memory Allgemein hat das Kopieren des globalen Speichers in den gemeinsamen Speicher eines Blockes mehrere Probleme mit sich geführt. Im Gegensatz zum Speicher einer C++-Anwendung ist die Betrachtung des Cuda-Speichers kompliziert. Zwar gibt es durch den Debugger einige Möglichkeiten sich diesen anzeigen zu lassen, jedoch sind insbesondere Indizierungsfehler äußerst schwer zu erkennen.

2 Phase 2: Implementierung Ähnlichkeitsmaße

2.1 Ansätze

In der Phase 2 haben wir versucht, direkt eine lauffähige und korrekte sequenzielle Lösung zu entwickeln. Diese Lösung haben wir direkt auf Cuda portiert, da Cuda ebenfalls für die zweite Phase am Vielversprechendsten erschien.

2.2 Sum of Squared Differences (SSD)

Für das *Block Matching* der beiden Bilder können verschiedene Abstandsmaße verwendet werden. Dank Epipolargeometrie müssen aber immer nur Pixel in der selben Zeile betrachtet werden. Außerdem betrachtet man zusätzlich einige um den jeweiligen Pixel herumliegende Pixel, um einen Pixel zu identifizieren. Der Abstand zwischen zwei Pixeln besteht daher, wenn man die *Sum of Squared Differences* verwendet, aus der Summe der quadrierten Abstände zwischen jeweils zwei Pixels im Betrachtungsradius. Das führt bei einer naiven Implementierung zu einem enormen Rechenaufwand. Aber es gibt einige Möglichkeiten den Rechenaufwand zu reduzieren.

2.2.1 Optimierungen

Cuda Zunächst haben wir die naive sequenzielle Implementierung naiv auf die Grafikkarte portiert. Das hat die Geschwindigkeit bereits um den Faktor 137 verbessert. Dies zeigt Tabelle 3.

Spalten Nach einiger Recherche fanden wir das Dokument von Joe Stam[1]. Stam weißt darin darauf hin, dass bei der naiven Variante mehrere Berechnungen redundant gemacht werden. Beispielweise werden alle Spalten mehrfach vergliechen. Das kann man vermeiden indem man die Kernel immer nur spaltenweise vergleicht und dann anschließend zusammenrechnet. Diesen Ansatz zeigt Abbildung 1 in Bild A. Außerdem lässt sich das sehr gut parallelisieren und auf mehrere Threads verteilen. Dies ist ebenfalls in Abbildung 1 Bild A ersichtlich. Für jeden Pixel der Bildbreite wird also ein eigener Thread angelegt. Das kann auch in mehreren Blöcken geschehen. Die Summen der Spalten werden dabei im *Shared Memory* gespeichert, was einen schnelleren Zugriff erlaubt.

Rolling Window Außerdem muss nicht wiederholt der komplette Kernel ausgerechnet werden, sondern nur die Änderungen berechnet werden. Das bedeutet, die oberste Zeile muss von der aktuellen Summe subtrahiert werden und die Zeile, die neu von dem Kernel überdeckt wird, muss hinzuaddiert werden. Durch diese Optimierung können wir die Rechenzeit (fast) unabhängig von der Kernelgröße halten, da die Kernelgröße nur für die obersten Zeilen relevant ist. Weiterhin ist die Rechenzeit linear zur Bildgröße. Das Prinzip des *Rolling Window* wird durch die Bilder B und D in Abbildung 1 verdeutlicht.

Die äußerer Schleife iteriert dabei über alle möglichen *Disparity*-Werte. Also von 0 bis *ndisp*. Die innere Schleife iteriert durch die einzelnen Spalten nach dem *Rolling Window*-Prinzip. Die jeweils besten Abstände und der dazugehörige *Disparity*-Wert wird dabei zwischengespeichert. Am Ende wird der beste Disparity-Wert und der Zwei-Beste noch auf die Bedingung

$$M_i * 1.05 < M_j, \forall i \neq j$$

überprüft und im Fall dass die Bedingung zutrifft M_i und andernfalls -1 in das jeweilige Pixel geschrieben.

Blockgröße Im Rahmen der letzten Wochen haben wir auch verschiedene Blockgröße auf unserem `ssd_cuda_v5` Algorithmus ausprobiert. Kleinere Blockgrößen haben dabei bessere Werte geliefert. Wir haben daher die Blockgröße auf 32 festgesetzt. Siehe dazu Tabelle 5 sowie Abbildung 4.

Kalkulation An dieser Stelle wollen wir einige Betrachtungen zum Rechenaufwand anstellen. Dazu betrachen wir die Pixelvergleiche. In der naiven Implementierung haben wir für jeden Pixel im Bild, den kompletten Kernel berechnen. Und das ganze *nDisp* mal. Eine Kernelberechnung besteht aus $2 \times kernelRadius + 1$ Vergleichen. Daraus ergibt sich folgende Formel.

$$(2 \times kernelRadius + 1)^2 \times width \times height \times ndisp$$

Die Performanz hängt hier also quadratisch von der Kernelgröße ab. In der optimierten Version werden nur noch einmalig die Spalten des Kernels berechnet. Diese summieren wir dann zusammen. Die Berechnung des Kernels beträgt daher $2 \times (2 \times kernelRadius + 1)$. Das Ganze machen wir *width* mal. Allerdings beginnen wir schon in der nächsten Reihen damit nicht den kompletten Kernel zu berechnen

sondern nur die Änderung. Die Änderung beträgt $2 \times width$. Wir kommen also auf folgende Formel:

$$(2 \times (2 \times kernelRadius + 1) \times width + 2 \times width \times (height - 1)) \times ndisp$$

Diese lineare Abhängigkeit vom Kernelradius bringt besonders bei großen Radien einen sehr großen Vorteil.

2.2.2 Weitere Optimierungsideen

Texturspeicher Als eine weitere Optimierung könnte man die Auswirkung der Texturspeicher untersuchen. Eventuell kann hier noch etwas an Performanz rausgeholt werden.

Shared Memory Außerdem könnte man noch einige Lesevorgänge aus dem globalen Speicher reduzieren, indem man den *Shared Memory* verwendet. Die Frage ist hierbei, ob sich der Overhead für das Laden in den *Shared Memory* lohnt.

2.3 ZNCC

Der Wertebereich des ZNCC Algorithmus liegt zwischen 0 und 1. Das bedeutet je größer der Rückgabewert ist, desto größer ist die Übereinstimmung der Bildausschnitte. Im Gegensatz zum SSD Algorithmus ist der ZNCC Algorithmus invariant gegen konstante Helligkeitsänderungen. Dadurch bieten sich mehrere Möglichkeiten zur Optimierung an, da die Standardabweichung und die Varianz für jeden Vergleich berechnet werden muss.

Nach der Implementierung einer sequenziellen Lösung war auch in diesem Filter ein Thread pro Pixel eine naheliegende Parallelisierung.

2.3.1 Optimierungen

Einteilung der Blöcke Die erste Überlegung die Cuda-Variante zu verbessern bestand darin, die Bilder zeilenweise zu bearbeiten. Ausgehend vom rechten Bild, berechnet jeder Thread genau einen Pixel. Jeder Block bearbeitet somit genau eine Zeile. Die Höhe gibt schließlich die Anzahl der Blöcke an.

Allgemein wird davon zunächst keine Performancesteigerung erreicht, es vereinfacht jedoch weitere Vorberechnungen. Durch die epipolargeometrischen Eigenschaften untersucht der Algorithmus nur zeilenweise die Bilder. Ein einzelner Pixel im rechten

Bild benötigt im linken Bild zur Berechnung mehr horizontale als vertikale Vergleichspixel. Die Anzahl der Überschneidungen mit einem Pixelnachbar in x-Richtung ist deshalb höher als mit einem Pixelnachbar in y-Richtung.

Dies ist besonders interessant für die Abspeicherung der Bilder im *shared memory*. Ein häufiges Wiederverwenden der Pixelwerte reduziert die Anzahl der lesenden Zugriffe im globalen Speicher. Auch ein vorberechneter Median und eine Standardabweichung kann zeilenweise wiederverwendet werden.

Einteilung der Blockdimensionen Problematisch an der *shared memory* Verwendung war die begrenzte Größe des verfügbaren Speichers. Einerseits wird dies durch die Gesamtgröße der jeweiligen Grafikkarte beschränkt. Andererseits führt eine zu hohe Verwendung dazu, dass die Auslastung der Multiprozessoren erheblich reduziert wird.

Die zuvor genannte Strategie, ein Thread für einen Pixel zu verwenden, stößt hier an seine Grenzen. Zudem hängt der notwendige Speicherplatz sowohl von der Kernelgröße als auch von *nDisp* ab. Der benötigte Speicherplatz für einen Block wird somit beschrieben durch:

$$(\#Threads + nDisp + windowSize - 1) \times windowSize$$

Version (`zncc_v3`) 3 nutzt jedoch 1024 Threads, was bei großer Kernelgröße schnell das Maximum an verfügbarem Speicher überschreitet. Schließlich war eine Umstrukturierung erforderlich. haben wir die Blockdimensionen genutzt, um auszudrücken, dass mehrere Threads die Kalkulationen für einen Pixel durchführen. Obwohl die Zwischenresultate dieser Threads zwischengespeichert werden müssen, sinkt insgesamt der Platzbedarf des Blockes.

Recheneinsparungen Da der Algorithmus nicht direkt den Wert des ZNCC-Algorithmus berechnet, sondern den Pixelabstand als Resultat liefert, können Einsparungen bei der Berechnung durchgeführt werden. Teile der Berechnung wie die abschließende Normalisierung haben auf den Vergleich der ZNCC-Werte keine Auswirkungen, sodass sie überflüssig ist. Dazu gehört auch die Standardabweichung im rechten Fenster. Auch hierbei handelt es sich um eine Division, die bei einem ZNCC-Vergleich beide Werte gleichermaßen beeinflusst.

Durch simple Formelumstellung können schließlich auch die Mittelwerte aus den inneren laufzeitkritischen Schleifen entfernt werden.

2.3.2 Auswertung

Tabelle 5 zeigt die Geschwindigkeiten der unterschiedlichen Implementierungen.

2.4 Allgemeine Schwierigkeiten

2.4.1 Cuda

Besonders bei Phase 2 war die Arbeit mit dem Cuda Debugger und Profiler sowohl wichtig als auch schwierig. Bugs, die die Indizierung von Speicher betreffen, sind trotz Debugger durch die hochgradige Parallelisierung schwer zu finden.

Gemeinhin hat der Profiler geholfen, potenzielle Optimierungen zu aufzuzeigen und die allgemeine Auslastung zu erkennen. Dennoch entsprachen die Unterschiede zwischen Windows und Linux nicht unseren Vorstellungen. Zwischenzeitlich lieferten die Profiler bei gleichem Quellcode verschiedene Resultate.

2.4.2 Probleme ZNCC

Register In einer Nvidia GPU stehen jedem Thread in einem Block mehrere Register zur Verfügung. Eine Überschreitung bestimmter Grenzwerte führt zu einer niedrigeren Auslastung. Trotz Reduzierung der Registeranzahl ist es uns nicht gelungen im ZNCC unter den angestrebten Grenzwert von 32 Registern zu gelangen.

Zwar kann man die Registeranzahl mit `-maxregcount` durch den Compiler beschränken, jedoch werden dabei die Register in den vergleichsweise inperformanten lokalen Speicher verschoben.

Strategien Im Vergleich zu den anderen Algorithmen haben wir lange Zeit benötigt, um Optimierungsstrategien zu finden. Nachdem wir die Vorberechnungen und den implementiert hatten und beide Bilder performant im *shared memory* abspeicherten, haben wir weitere Ansätze gesucht. Im Gegensatz zum Bilateral-Filter hat der Profiler wenig zu dieser Suche beigetragen.

Die längste Zeit sind wir davon ausgegangen, dass das im SSD beschriebene Verfahren nicht auf den ZNCC anwendbar ist. Erst die oben genannte Formelumstellung hat dies kurzfristig offenbart, womit sicher noch eine weitere Optimierung möglich ist.

Anhang

Git Tag	Algo/Optimierung	Zeit [ms]
gauss_v1	Sequenziell	3887
gauss_v2	Naive Cuda Implementierung	219
gauss_v2	Separiert	3

Tabelle 1: Performanzdaten *BM_G16* des Gaussfilters

Git Tag	Algo/Optimierung	Zeit [ms]
bilateral_v1	Sequenziell	113409
bilateral_v2	Naive Cuda Implementierung	35848
bilateral_v3	Kernel vorberechnet und komprimiert	1083
bilateral_v4	Separiert und Shared Memory	76
bilateral_v5	Double zu Float	54
bilateral_v6	Pixelbreite angepasst	20

Tabelle 2: Performanzdaten *BM_BIG_B6_300* des Bilateralfilters

Git Tag	Algo/Optimierung	Zeit [ms] SSD15	Zeit [ms] SSD20
znccv1	Sequenziell	44404	NA
ssd_v2	(second)bestValue als float anstatt double	44033	NA
ssd_cuda_v2	Naive Cuda Implementierung	320	43895
ssd_cuda_v3	Berechne immer nur Spalten und summiere diese zusammen	1433	15758
ssd_cuda_v4	Ränder nicht explizit auf -1 setzen	1141	12644
ssd_cuda_v5	Mit Rolling Window	119	1287

Tabelle 3: Performanzdaten des SSD Algorithmus

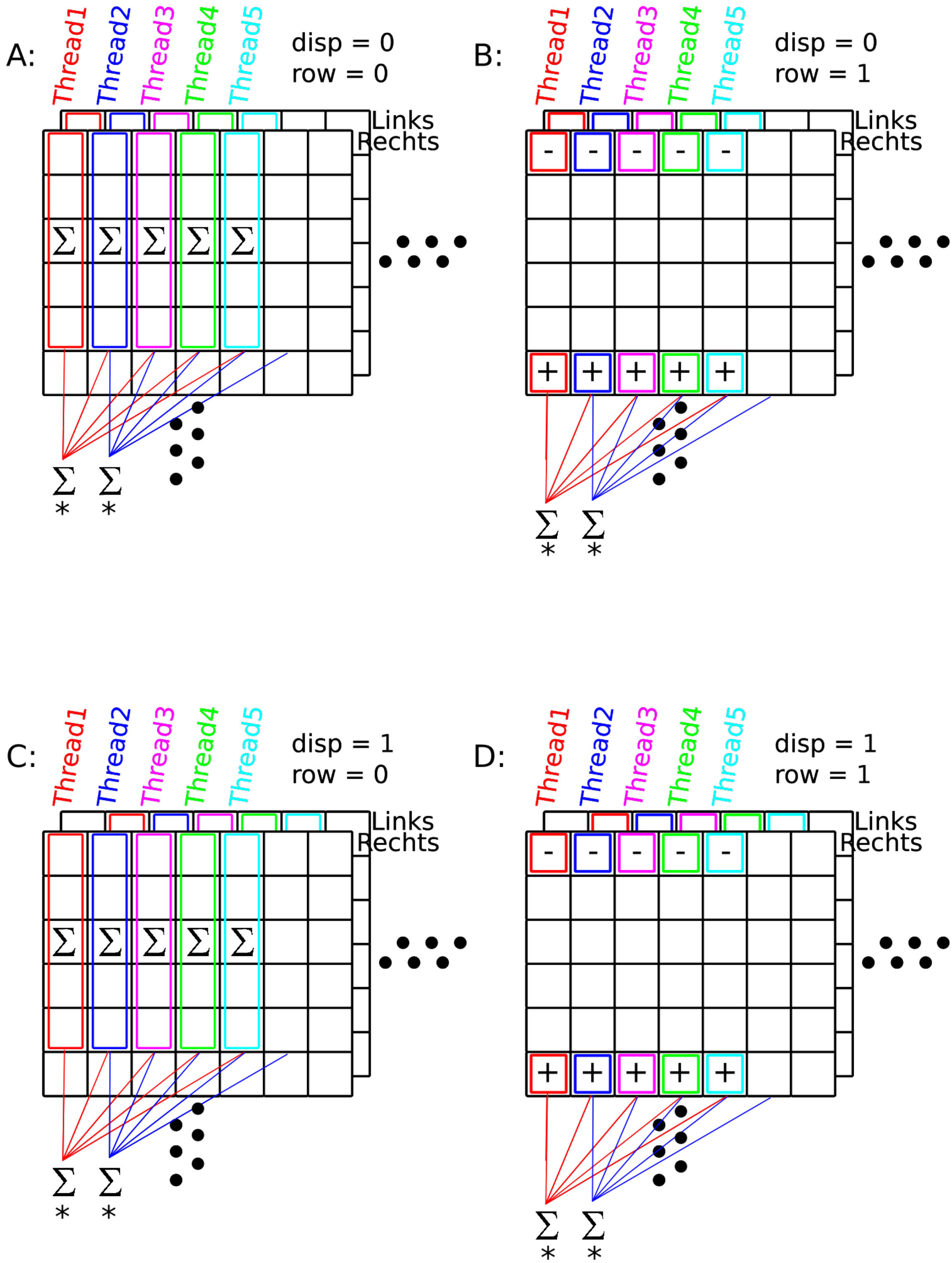

Abbildung 1: Visualisierung des SSD Algorithmus

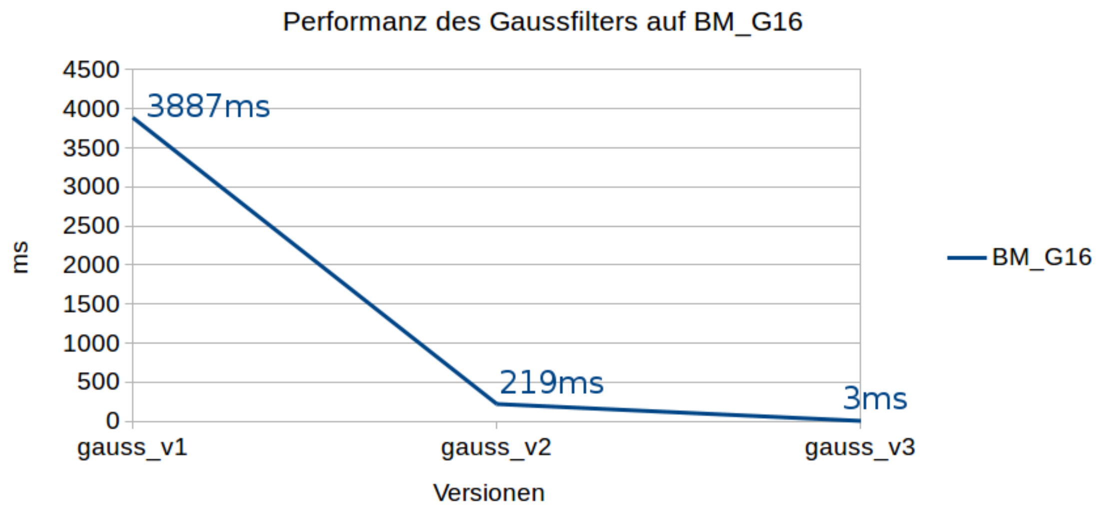

Abbildung 2: Performanz des Gaussfilters (BM_G16)

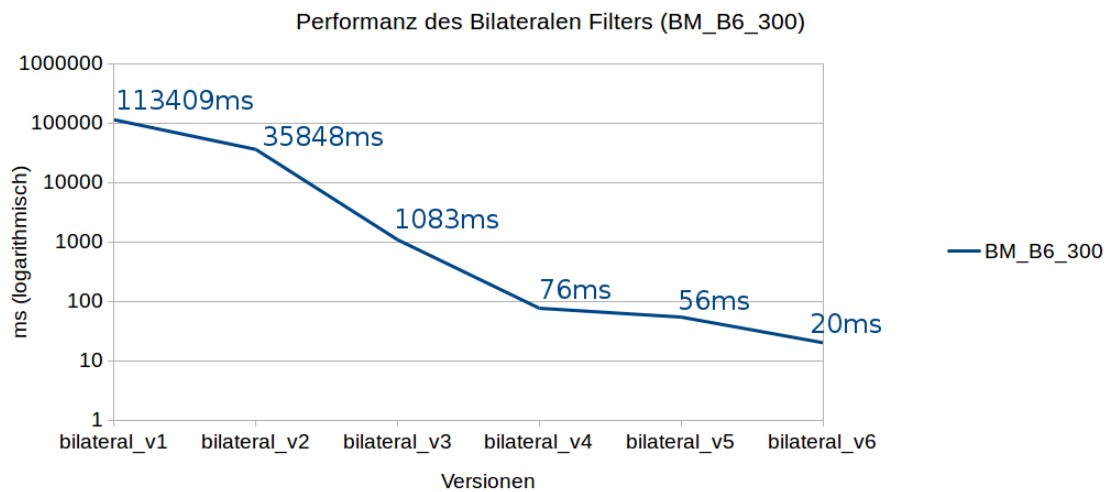

Abbildung 3: Performanz des Bilateralenfilters (BM_B6_300)

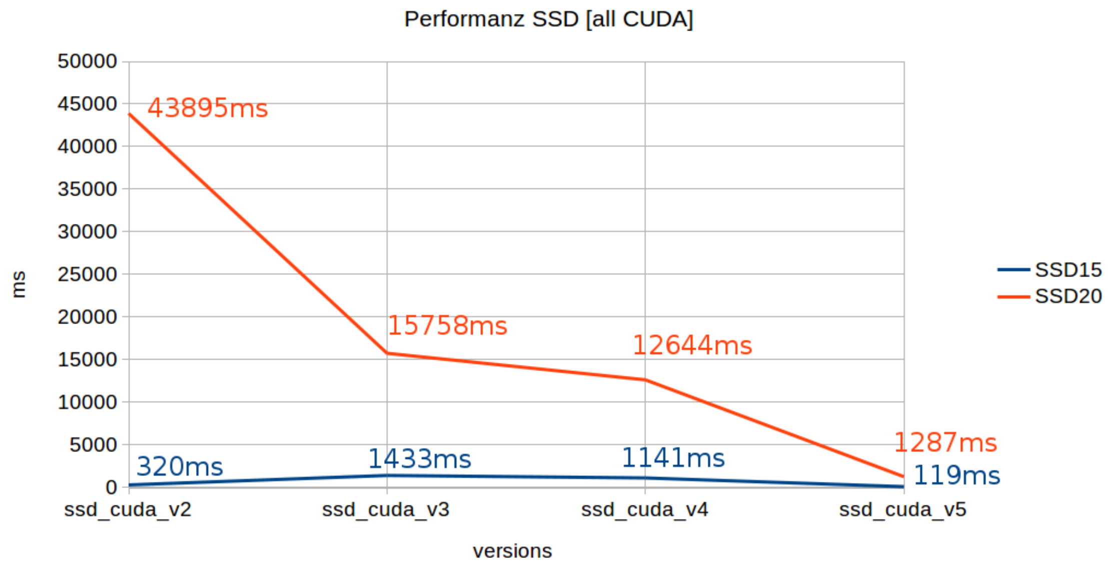

Abbildung 4: Performanz des SSD Algorithmus

Blockgröße	**Zeit [ms]**
32	1231
64	1274
128	1288
256	1300

Tabelle 4: Performanzdaten unterschiedlicher Blockgrößen des ssd_cuda_v5 auf *BM_SSD20 real_time*

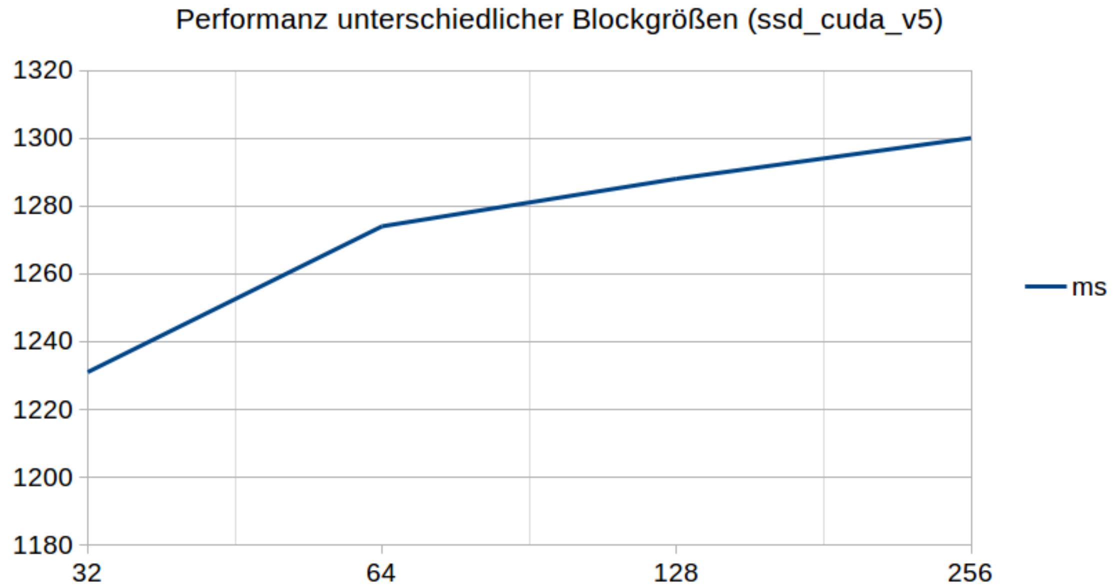

Abbildung 5: Performanz unterschiedlicher Blockgrößen (ssd_cuda_v5)

Git Tag	**Algo/Optimierung**	**Zeit [ms] ZNCC15**	**Zeit [ms] ZNCC20**
zncc_v1	Naiv, sequenziell	91268	NA
zncc_v2	Naive Cuda Implementierung	NA	NA
zncc_v3	Vorberechnung des Medians (linkes Bild)	295	37149
zncc_v4	Mehrere Threads pro Pixel	207	30564
zncc_v5	Speichern der benötigten Bildteile im *shared memory*	144	20601
zncc_v6	Vorberechnung des Medians nach Umstrukturierung	98	13284

Tabelle 5: Performanzdaten des ZNCC Algorithmus

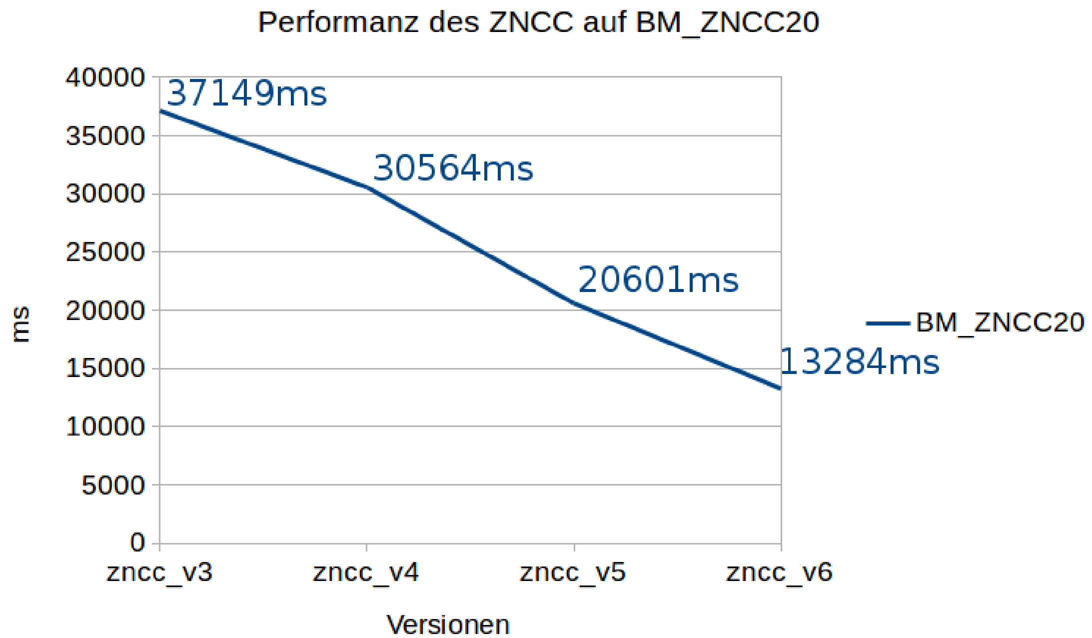

Abbildung 6: Performanz des ZNCC Algorithmus

Literatur

[1] Joe Stam. Stereo Imaging with CUDA. Online erhältlich unter `goo.gl/gLtmQY`; abgerufen am 21. Januar 2018., 2008.