Fachhochschule Köln
Cologne University of Applied Sciences
Fakultät für Informatik und Ingenieurwissenschaften

Optimierung eines neuen Logarithmic-Search-Verfahrens zum Image Mosaicing unter Einsatz des CUDA-Frameworks

Masterarbeit
zur Erlangung des akademischen Grades
„Master of Science in Media Informatics“

Vorgelegt von: Eugen Sewergin
Matrikelnummer: 11042216

Erstprüfer: Prof. Dr. Wolfgang Konen
Zweitprüfer: Prof. Dr. Horst Stenzel

Vorgelegt am: 07.04.2009

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Einleitung

Bei endoskopisch unterstützten Operationen müssen sich Chirurgen mit einer klassischen beschränkten Darstellung[1] der Endoskopkamera befassen. Hierbei können Orientierungsstörungen auftreten. Eine Abhilfe bei solchen Fällen bietet das *Image Mosaicing* an, das ein Übersichtsbild aus mehreren Bildern einer Bildfolge erstellt. Dem operierenden Chirurg wird dadurch zusätzlich zum Kamerabild des Endoskops eine erweiterte Übersicht geboten, um die Orientierung nicht zu verlieren und seine Arbeit zu erleichtern.

Im Folgenden wird zuerst die Motivation, sowie der Ausgangspunkt der Arbeit geschildert. Im nachkommenden Abschnitt werden die Ziele der Masterarbeit vorgestellt. Anschließend geschieht eine Einordnung der Masterthese in den allgemeinen Stand der Forschung. Zu guter letzt schließt das Kapitel mit dem Aufbau der Arbeit ab.

1.1 Motivation

Aus der Sicht der computergestützten Bildverarbeitung wurde bisher keine Aufmerksamkeit bezüglich der Auslagerung von Berechnungen, rechenintensiver Image Mosaicing-Verfahren, von der zentralen Recheneinheit eines Computers (engl. Central Processing Unit, folglich CPU) auf die zentrale Recheneinheit einer Grafikkarte (engl. Graphics Processing Unit, folglich GPU) geschenkt.

Die Stärke der GPUs ist in den letzten Jahren drastisch gestiegen[2] und das Ende des Leistungszuwachses wurde noch lange nicht erreicht.

Aus diesem Grund wäre es wünschenswert rechenintensive Berechnungen auf die GPU auszulagern und somit die CPU zu entlasten. Daraus folgernd stände die CPU für andere

[1] So genannter Scheuklappeneffekt
[2] Siehe Abbildung 2.1

Berechnungen frei, sodass eine Performance-Steigerung (Verringerung der Rechenzeit) bei einem Image Mosaicing-Verfahren zu erzielen wäre.

1.2 Ausgangspunkt

Das ImageJ[3]-Plugin zum Image Mosaicing von Naderi[4], Zimmermann[5] und weiteren Beteiligten, das in dem Forschungsprojekt „Bildverarbeitung und 3D-Navigation in der Endoskopie“[6] entwickelt wurde, ist der Ausgangspunkt dieser Arbeit.

Das Plugin wurde von Naderi in Java als echtzeitfähige Anwendung implementiert und auf Performance optimiert. Eine Ergänzung des Plugins um eine erweiterte Benutzeroberfläche und verschiedenen Möglichkeiten zum Input und Output erfolgte in der Arbeit von Zimmermann.

Im Kern beruht die bestehende echtzeitfähige Implementierung des algorithmischen Kerns in Java auf dem Optical Flow-Verfahren nach Kourogi[7].

Mit Hilfe des Plugins lässt sich aus mehreren Bildern einer endoskopischen Bildsequenz ein Übersichtsbild (Panoramabild) erstellen. Hierzu können entweder Bilder aus einer Live-Videosequenz eines Endoskops oder aus einer lokal angelegten Bildsequenz verwendet werden. Es werden immer zwei nacheinander folgende Bilder im Bitmap-Format dem intensitätsbasierten Algorithmus-Verfahren zur Verarbeitung gegeben. Wobei hier geprüft wird, ob der Farbraum der Bilder als 8 Bit Grauwert oder als RGB vorliegt. Im ersten Fall wird mit den originalen Grauwertbildern gearbeitet und auf diese bei der Erstellung des Panoramabildes zurückgegriffen. Im zweiten Fall werden zuerst Kopien der Farbbilder angelegt, die vor ihrer Verwendung in Grauwertbilder umgerechnet werden. Bei der Erzeugung des Farbpanoramabildes werden wiederum Farbbilder verwendet.

Mit einfachen Bildsequenzen arbeitet das Plugin erfolgreich, hat aber auf realen Endoskopvideosequenzen mit stark variierender Beleuchtung noch erhebliche Probleme. Ein kürzlich entwickelter und in das Plugin implementierter neuer Algorithmus auf Basis von Kreuzkorrelationskoeffizienten und Logarithmic-Search zeigt eine höhere stabilität.

[3] ImageJ ist ein Java-basiertes Bildverarbeitungsprogramm, das primär in der medizinischen und biologischen Bildverarbeitung eingesetzt wird.
[4] [Naderi07]
[5] [Zimm08]
[6] `http://www.gm.fh-koeln.de/~konen/Diplom+Projekte/FProjekt-BV-3D-Endo/FProjekt-BV-3D-Endo.htm`
[7] [Kourogi99]

Allerdings muss er noch optimiert werden, was die Qualität und die Performance angeht.

1.3 Zielsetzung

Ziel dieser Arbeit ist eine Optimierung des bereits in das Plugin implementierten neuen Logarithmic-Search-Verfahrens zum Image Mosaicing. Zum einen soll eine Performancesteigerung, also eine Verringerung der Rechenzeit, erzielt werden und somit die Echtzeitfähigkeit des Plugins steigern. Zum anderen sollen die Algorithmus-Parameter und die algorithmische Qualität optimiert werden.

Des weiteren soll der wissenschaftlichen Frage „Wie gut gelingt es, typische Algorithmen der Bildverarbeitung auf ein paralleles Framework abzubilden?“ nachgegangen werden. Dazu soll ein Konzept zur Auslagerung rechenintensiver Teile des Algorithmus auf CUDA-Framework von NVIDIA entwickelt und umgesetzt werden.

1.4 Einordnung der Arbeit

Diese Masterarbeit entstand im Rahmen des Forschungsprojektes „Bildverarbeitung und 3D-Navigation in der Endoskopie“ mit dem Schwerpunkt Image Mosaicing, das vom Prof. Dr. Wolfgang Konen[8] in Zusammenarbeit mit PD Dr. med. Martin Scholz[9] durchgeführt wird.

Da die vorliegende Arbeit den Bereich des Image Mosaicing auf endoskopischen Bild- bzw. Videosequenzen beinhaltet, darüber hinaus die parallele Verarbeitung von Algorithmen auf GPUs aufgreift, wird im Folgenden kurz auf den Stand der Forschung in den zutreffenden Bereichen eingegangen.

Im Kontext des Image Mosaicing sind viele feature- und intensitätsbasierte Algorithmen anzutreffen. So existieren Veröffentlichungen[10] zu feature-basierten Verfahren, die durch eine Extraktion und Matching von bestimmten Merkmallen (Detecting Points of Interest[11]) wie z.B. Kanten ein Übersichtsbild erstellen. Im Gegensaz zu feature-basierten Verfahren wird beim Image Mosaicing mit intensitätsbasierten Methoden[12]

[8] Professor für Angewandte Informatik und Mathematik an der FH Köln
[9] Leitender Oberarzt der Neurochirurgischen Klinik an der Ruhr-Universität Bochum
[10] Beispielsweise [SPM02], [ZFD97]
[11] Erkennung und Erfassung von bestimmten Punkten innerhalb eines Bildes
[12] Beispielsweise [SzSh97], [Kourogi99]

auf die Heligkeitsinformationen eines Überlappungsbereichs zweier konsekutiver Bilder zurückgegriffen.

Ein Survey von Flusser at al. bietet eine Übersicht über verschiedene Image Mosaicing-Verfahren[13].

Im Bereich der Endoskopie finden sich Arbeiten[14], die sich der computergestützten Bildverarbeitung bedienen, um Bildverbesserungen in endoskopischen Videosequenzen zu erzielen. Des weiteren gibt es Veröffentlichungen[15], welche sich mit der automatischen Erstellung von Übersichtsbildern aus endoskopischem Bildmaterial auseinandersetzen.

Die Leistungsfähigkeit der GPUs ist für viele Wissenschaftler aber auch Unternehmen interessant. So finden sich in dem Bereich Arbeiten zu Berechnungen der physikalischen Simulationen[16], effiziente Lösung von Sparse Matrizen auf GPUs [17], Bildanalyse und Bildverarbeitung[18], Lineare Algebra Implementierungen auf Grafikkarten[19], Mittels GPU beschleunigte Algorithmen wie: Discrete Euclidean Distance Transformation[20], Algorithmen aus der medizinischen Bildverarbeitung[21], Sortieralgorithmen[22].

Eine ausführliche Übersicht im Bereich *General-Purpose computation on GPUs* (GPGPU) bietet ein Survey von Owens at al.[23].

Das Thema der Berechnung von allgemeinen Aufgaben auf dem Grafikprozessor, bildet inzwischen eine rege Gemeinschaft aus verschiedenen Forschern und Wissenschaftlern[24].

Zunehmend entwickeln auch Unternehmen Consumer-Lösungen in den rechenintensiven Bereichen Video-, Audio- und Bildverarbeitung, die sich der Rechenleistung der GPU bedienen. So wandelt beispielsweise das Programm *BadaBoom* von Elementals mit Hilfe der GPU einen zwei stündigen HD-Film ins iPod-Format bis zu 18x schneller um, als eine Quad-Core-CPU von Intel[25]. Ein weiteres Beispiel stellt *Common Vision Blox* von Stemmer Imaging dar, das Teile der Verarbeitung auf die Grafikkarte des PC-Systems

[13] [ZF03]
[14] Beispielsweise [FVLS04], [VKNS03]
[15] Beispielsweise [Behrens08], [KBS07]
[16] [FWKL05]
[17] [BFGS03]
[18] [CBDR03], [MDP06]
[19] [KW03]
[20] [JKW09]
[21] [Schiwietz08]
[22] Beispielsweise [GRHTM05], [GGZ06]
[23] [OLGHKLP07]
[24] `http://gpgpu.org/`
[25] [Netzwelt08]

auslagert, um Geschwindigkeitsteigerungen bei der Bildverarbeitung zu erzielen[26]. Adobe hat bereits eine GPU unterstützte Beschleunigung in *Adobe Reader* sowie *Adobe Acrobat* implementiert[27] und plant die Nutzung der GPU für rechenintensive Berechnungen in *Adobe Photoshop*[28].

1.5 Aufbau der Arbeit

Im nachfolgenden Kapitel werden zunächst notwendige Grundlagen, die zum Verständnis der Arbeit dienen, behandelt. Das dritte Kapitel enthält zum einen alle Schritte zur Optimierung der Performance des Logarithmic-Search-Verfahrens auf der CPU, zum anderen eine Betrachtung einiger Parameter, die sich auf die Qualität des Panoramabildes auswirken. Wie sich CUDA in Java benutzen lässt, stellt das vierte Kapitel dar. Anschließend folgt im nächsten Kapitel die Vorstellung eines Konzeptes zur Portierung rechenintensiver Teile des ImageJ-Plugins in CUDA. Auf die Umsetzung des Konzeptes wird im sechsten Kapitel detailliert eingegangen. Daraufhin erfolgt im siebten Kapitel die Gegenüberstellung und Auswertung der CPU- und GPU-Ergebnisse. Abschließend fasst das letzte Kapitel die Ergebnisse der Arbeit zusammen und liefert einen Ausblick auf mögliche Erweiterungen zur Performance- und Qualitätssteigerung des Logarithmic-Search-Verfahrens.

[26] [StemmerI07]
[27] [Adobe08]
[28] [Golem08]

Kapitel 2

Grundlagen

Nachfolgend werden Grundlagen, die für das Verständnis dieser Arbeit erforderlich sind, angesprochen. So erfolgt zunächst eine Übersicht zum CUDA-Framework. Daraufhin werden Verfahren im Zusammenhang mit dem Logarithmic-Search-Verfahren des ImageJ-Plugins vorgestellt. Dazu gehört das Optical Flow-Verfahren nach Kourogi, die Suchstrategie Logarithmic-Search und ein Maß zur Bestimmung der Differenz zwischen zweidimensionalen Bildfunktionen.

2.1 CUDA-Framework

Die Rechenleistung der Grafikkarten im Vergleich zu Desktopprozessoren wuchs im Zeitraum von 2003 bis 2008 bei der Fließkommaberechnung pro Sekunde um ein Vielfaches an. Dies Verdeutlicht die Abbildung 2.1.

Die Wachstumsrate mit einem Faktor von 1,7 bis 2,3 pro Jahr übersteigt das Moorische Gesetz (Faktor 1,4 pro Jahr), das bei den CPUs verwendet wird[1]. Der Grund für die Diskrepanz in der Fließkommaberechnung zwischen GPUs und CPUs sind verschiedene Architekturen, die in der Abbildung 2.2 schematisch illustriert sind. So werden bei den GPUs von NVIDIA mehr Transistoren der Datenverarbeitung gewidmet als dem Cache und den Kontroll-Einheiten.

Um die enorme Rechenleistung der Recheneinheiten einer Grafikkarte zu nutzen, entwickelte NVIDIA Anfang 2007 das so genannte „Compute Unified Device Architecture“ (CUDA), das mittlerweile in der Version 2.0 kostenlos zu beziehen ist[2].

[1] [LOGKHLT05]
[2] http://www.nvidia.com/object/cuda_get.html

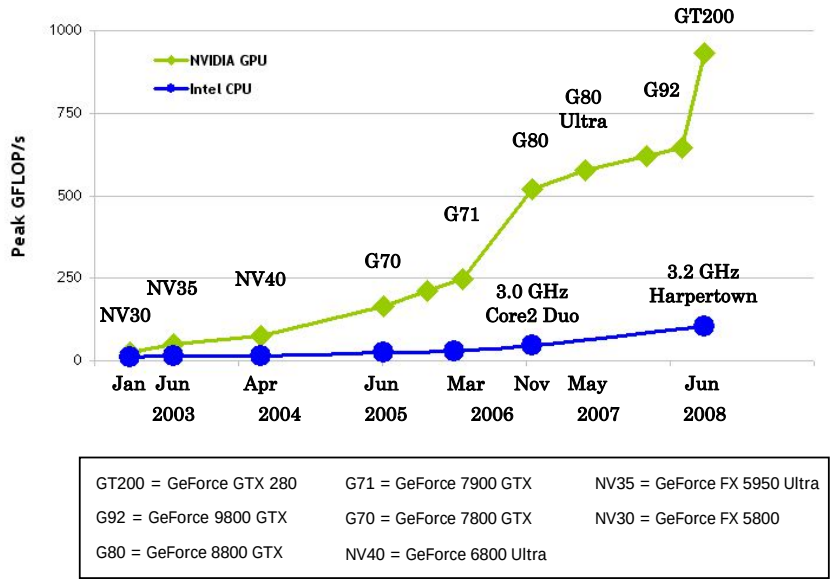


Abbildung 2.1: Rechenleistung bei Fließkommaberechnungen von NVIDIA GPUs zu Intels CPUs im direkten Vergleich [CUDAPG08]

Aufgrund einer Erweiterung der Programmiersprache C durch zusätzliche Sprachkonstrukte, bietet CUDA eine high-level Programmierung für Grafikkarten der NVIDIA-Produktfamilie ab der Geforce 8000 an. Einem Entwickler wird damit geboten ein Programm zu schreiben, das datenparallele Teile auf der GPU und den Rest des Programms auf der CPU ausführt. Somit lässt sich die Grafikkarte als Co-Prozessor für parallele Verarbeitung verwenden.

Den GPGPU-Markt erkannten auch AMD und Intel. Seit Mitte 2008 bietet AMD so genannte *ATI FireStream Processors* mit einem *ATI Stream SDK* an, das auf der Programmiersprache Brooks+ aufbaut[3]. Intel arbeitet gerade an dem Projekt *Larrabee*, das Ende 2009 bzw. Anfang 2010 erscheinen und als ein Multiprozessorsystem sich in dem GPGPU-Markt etablieren soll[4].

[3] [AMD08]
[4] [HT4U08]

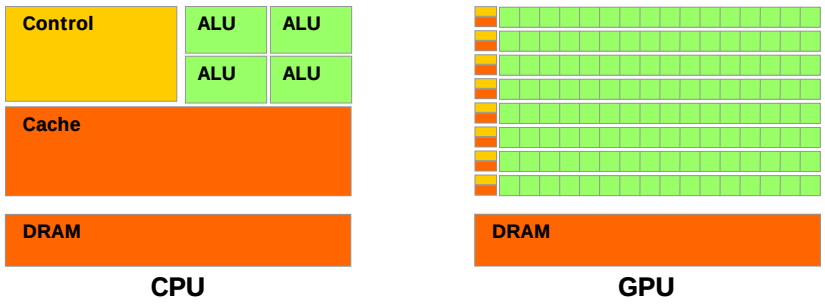


Abbildung 2.2: Schematische Architektur einer 4-Kern-CPU und einer Multiprozessor-GPU [CUDAPG08]

2.1.1 Hardware Modell

Eine Grafikkarte von NVIDIA wird im CUDA-Kontext als *Device* und deren Arbeitsspeicher als *Device Memory* bezeichnet[5]. Analog dazu ist *Host* eine CPU und *Host Memory* der Arbeitsspeicher.

Device Memory lässt sich in folgende drei Bereiche einteilen:

1. **Constant Memory** - Ein 64 KByte großer Speicherbereich ist im Arbeitsspeicher der Grafikkarte für Constant Memory reserviert. Hier werden konstante Daten gehalten, die von der GPU nicht zur Laufzeit verändert werden dürfen.

2. **Texture Memory** - Ein als Textur allozierter Speicher wird im Texture Memory gehalten. Der Texturspeicher bietet hardwaremäßige Unterstützung für 1D, 2D und 3D Texturen.

3. **Global Memory** - Alle weiteren allozierte Speicher werden automatisch im Global Memory angelegt. Wobei die Größe der beiden zuletzt genannten Speicherbereiche jeweils vom Grafikkartenmodell abhängig ist.

Die Latenzzeit beim Zugriff je Speicher liegt bei 400 bis 600 Takte. Wobei der Speicherzugriff auf Constant Memory und Texture Memory gepuffert ist und im Falle eines Cache-Hit[6] einen Takt beträgt.

[5] Siehe Abbildung 2.3
[6] Die aus dem Speicher zu holende Daten befinden sich im Cache.

Device besteht aus einem Set von mehreren Multiprozessoren (MP). Jeder der MPs beinhaltet acht Prozessorkerne[7], die parallel zu einander die selbe Instruktion auf verschiedenen Daten ausführen. Somit handelt es sich um eine SIMD-Architektur (*Single Instruction Multiple Data*). Weiterhin sind innerhalb eines Multiprozessors Speicherressourcen für die Datenverarbeitung verfügbar. Das sind im Einzelnen:

1. **Shared Memory** - Jeder MP verfügt über 16 KByte Shared Memory, das in 16 32-Bit *Banks* aufgeteilt ist und von jedem der acht Kerne parallel beschrieben und gelesen werden kann. Die Latenzzeit der Lese- und Schreibzugriffe beträgt einem Takt solange keine Bank-Konflikte[8] entstehen.

2. **Register** - Pro MP sind 8192 32-Bit Register je 1024 Register pro Prozessorkern verfügbar. Zugriffszeiten eines Registers betragen ebenfalls einem Takt.

3. **Texture Cache** - Mit der Größe von 16KByte hält der Texture Cache die zuvor aufgeforderten Daten aus dem Texturspeicher (Texture Memory). Im Falle eines Cache-Hit beträgt die Zugriffszeit einem Takt. Bei Cache-Miss liegt die Latenzzeit zwischen 400 und 600 Takten.

4. **Constant Cache** - Der Constant Cache, 8 KByte groß, arbeitet ebenfalls als Zwischenspeicher und hält bereits verwendete Daten aus dem Constant Memory. Zugriffszeiten bei Cache-Hit und Cache-Miss sind gleich den Zugriffszeiten des Texture Cache.

Wie aus der Abbildung 2.3 ersichtlich, kann jeder Multiprozessor auf Texture Memory und Constant Memory nur lesend zugreifen. Beim Global Memory, Shared Memory und den Registern kann der Lese- und Schreibzugriff beliebig lokalisiert sein. Damit ist *Scatering* und *Gathering* möglich[9].

Die Kommunikation zwischen dem Device und Host erfolgt über Texture Memory, Global Memory und Constant Memory. Ein Host hat dabei den vollen Lese- und Schreibzugriff auf die genannten Speicher.

[7] Auch als Scalar Processor oder Stream Processor bezeichnet.
[8] Ein Bank-Konflikt entsteht wenn auf eine Bank gleichzeitig parallel zugegriffen wird. In dem Fall werden die Zugriffe serialisiert.
[9] Nähere Informationen sind dem [CUDAPG08] zu entnehmen.

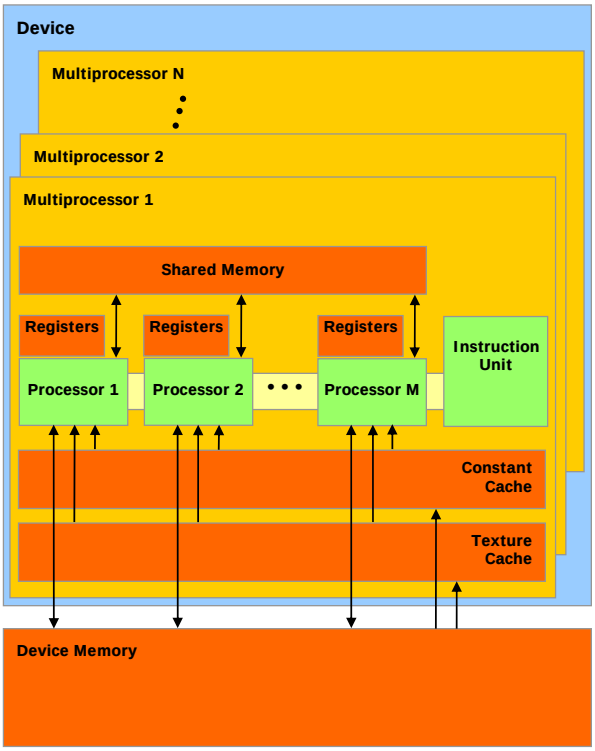


Abbildung 2.3: Schematischer Aufbau einer Grafikkarte von NVIDIA im Kontext von CUDA [CUDAPG08]

2.1.2 Software Modell

Ein Programm im CUDA-Kontext besteht aus einem C-Code, der auf der CPU (*Host*) sequenziell ausgeführt wird und einem CUDA-spezifisch erweiterten C-Code, dem so genannten *Kernel*, der auf der GPU (*Device*) zur Ausführung kommt.

Ein Kernel wird immer für ein spezielles Betriebssystem mit dem im CUDA-Toolkit mitgelieferten NVCC-Compiler vorkompiliert, nicht jedoch für ein spezielles Grafikkarten-Modell. Stattdessen wird er zur Laufzeit von dem Grafikkartentreiber in eine Maschienensprache für eine vorhandene Grafikkarte übersetzt und ausgeführt. Damit wird gewährleistet, dass einmal geschriebener Kernel auch auf zukünftigen CUDA-fähigen Grafikkarten ausgeführt werden kann.

Im Kernel werden Instruktionen für so genannte *Threads* vorgeschrieben. Alle Threads sind in Blöcke unterteilt. Diese befinden sich wiederum in einem *Grid*. Abbildung 2.4

verdeutlicht den beschriebenen Zusammenhang.

Die Konfiguration eines Kernels wird von einem Programmierer übernommen. Er legt die Grid-Dimension und die Block-Dimension fest. Die Konfigurationsparameter sind im Einzelnen:

- **Grid** - vom Typ dim3[10]. Legt die Dimensionen und ihre Größen fest. Ein Grid darf eindimensional oder zweidimensional deklariert werden. Dabei wird seine Größe in Blöcken angegeben. Maximal dürfen 65535 Blöcke pro Dimension verwendet werden.

- **Block** - ebenfalls vom Typ dim3. Im Gegensatz zu einem Grid kann ein Block bis zu drei Dimensionen haben. Die Größe eines Blocks wird in Threads festgelegt, wobei diese nicht 512 Threads pro Block übersteigen darf.

- **Memsize** - vom Typ size_t, gibt pro Block den dynamisch zugeteilten Shared Memory in Byte an. Maximal können bis zu 16 KByte verwendet werden.

Code Beispiel 2.1 zeigt einen Kernel *increment_gpu* und seine Konfiguration beim Aufruf. Dieser inkrementiert parallel auf einer GPU ein Float-Array um einen Wert. Das Schlüsselwort *__global__* gibt an, das es sich um einen Kernel handelt.

Die Konfigurationsparameter selber können in einem Kernel durch die speziellen Variablen *gridDim* und *blockDim* abgefragt werden. Weiterhin hat jeder Thread einen Zugriff auf seine Grid- und Block-Koordinaten durch die Variablen *blockIdx* und *threadIdx*. Mit Hilfe dieser lassen sich Iterationsvariablen erzeugen (Zeile 5). Der Aufruf des Kernels erfolgt in Zeile 19. Dieser bekommt die Konfigurationsparameter und Variablen als Aufrufparameter übergeben.

Während der Ausführung eines Kernels werden die Blöcke eines Grids automatisch auf die Multiprozessoren verteilt und die Threads eines Blocks in *Warps* gruppiert. Ein Warp beinhaltet 32 konsekutive Threads eines Blocks. Das heißt, dass die ersten 32 Threads den ersten Warp, die nächsten 32 Threads den zweiten Warp, usw. darstellen. Besteht also ein Block aus n Threads, so gibt es $n/32$ Warps pro Block, wobei der letzte Warp mit wirkungslosen Threads aufgefüllt wird, falls dieser nicht voll ist. Die Warps werden in CUDA-Code nicht explizit deklariert. Ihre Kenntnis kann aber zum effizienten und optimierten Programmieren genutzt werden.

[10] CUDA-spezifischer Datentyp

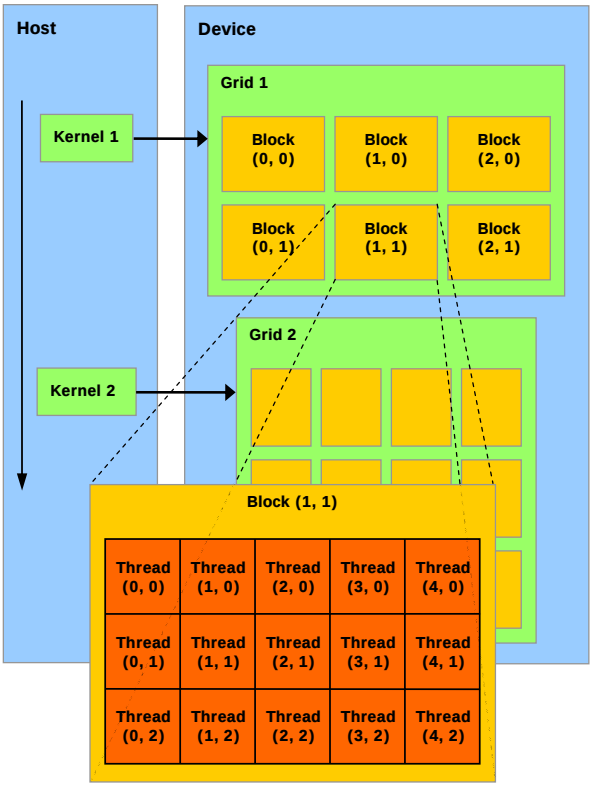


Abbildung 2.4: Schematischer Aufbau eines Kernels in CUDA [CUDAPG08]

Pro Multiprozessor wird ein Warp eines Blocks in vier Schritten abgearbeitet. Dabei arbeiten die acht Prozessorkerne eines MPs acht Threads parallel pro Schritt ab, falls es zu keiner Divergenz in einem Warp gekommen ist. Eine Divergenz, z.B. jeder zweite Thread eines Warps geht einen anderen Weg im Kernel, wird vor der Ausführung sequenzialisiert. Würde also jeder Thread eines Warps einen anderen Weg im Kernel einschlagen, stiege damit die Ausführungszeit durch die Sequenzialisierung um das 32-fache.

Eine explizite Synchronisations-Barriere für alle Threads eines Blocks im Kernel stellt die CUDA-Funktion *__syncthreads* dar. Mit ihrer Hilfe lässt sich ein Punkt im Kernel markieren. An diesem Punkt müssen alle Threads eines Blocks angekommen sein, bevor es mit der parallelen Instruktionsausführung weiter geht. Genutzt wird die Synchro-

Listing 2.1: Beispielcode für eine parallele Inkrementierung eines Float-Arrays um einen Float-Wert auf der GPU

```
1  // Kernel Deklaration
2  __global__ void increment_gpu(float *a, float b, int N)
3  {
4          //Erzeuung einer Iterationsvariable idx
5          int idx = blockIdx.x * blockDim.x + threadIdx.x;
6
7          if( idx < N)
8          a[idx] = a[idx] + b;
9  }
10 ...
11 void main()
12 {
13 ...
14         //Eindimensionale Thread-Anzahl im Block festlegen
15         dim3 dimBlock( blocksize, 0, 0 );
16         //Eindimensionale Block-Anzahl im Grid festlegen
17         dim3 dimGrid( N / blocksize, 0, 0 );
18         // Kernel Aufruf
19         increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
20 ...
21 }
```

nisation bei datenabhängigen Lese- und Schreibzugriffen[11], um Wettlaufsituationen zu vermeiden.

2.1.3 Speicher Modell

Eine schematische Darstellung verschiedener Interaktionsmöglichkeiten eines Threads mit allen Speicherressourcen einer Grafikkarte zeigt die Abbildung 2.5. Ein Thread kann demnach lesend und schreibend auf Register, Local Memory[12], Shared Memory, sowie Global Memory zugreifen. Speziell über Shared Memory lässt sich ein Datenaustausch zwischen Threads eines Blocks realisieren. Ein Tausch von Daten zwischen den Blöcken kann über Global Memory erfolgen. Weiterhin bieten Constant Memory und Texture Memory einen gepufferten Lesezugriff für Threads jedes Blocks an.

Aufgrund der unterschiedlichen Latenzzeiten von Speicherressourcen, zusehen in der

[11] Read after Write (RAW), Write after Read (WAR) und Write after Write (WAW).
[12] Local Memory wird bei der Auslagerung von nicht in Register passenden Daten verwendet (*spilling*).

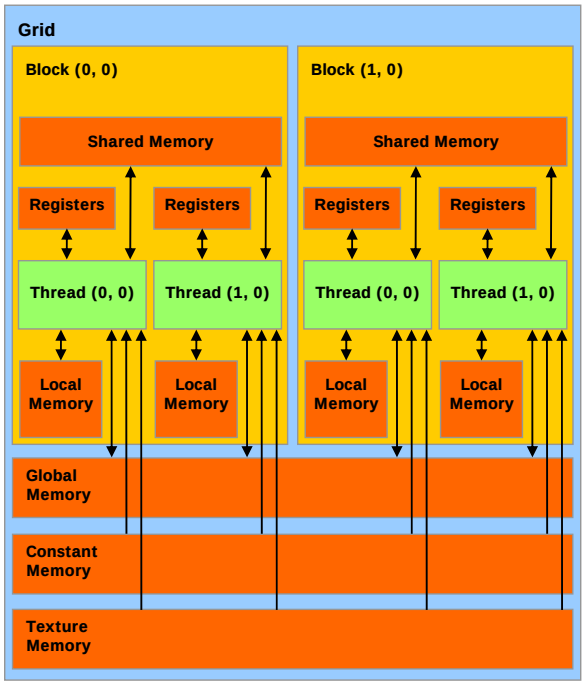


Abbildung 2.5: Interaktionsschema zwischen Threads und Speicherressourcen in CUDA [CUDAPG08]

Tabelle 2.1, kristallisiert sich ein allgemeines Muster bei der CUDA-Programmierung in der Praxis[13]:

1. Lesen der Daten aus Global Memory, Constant Memory bzw. Texture Memory vor Berechnungen in den schnellen Shared Memory.
2. Durchführung von Lese- und Schreibzugriffen bei Berechnungen auf dem Shared Memory.
3. Schreiben der Resultate aus dem Shared Memory in den Global Memory.

2.1.4 Scheduler

Eine optimale Aufteilung der Rechenarbeit wird von einem Scheduler unternommen. Sein Ziel ist es, eine möglichst durchgängige Nutzung aller zur Verfügung stehenden

[13] http://www.nvidia.com/object/cuda_home.html

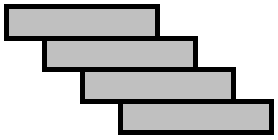
Speicher	Ort	Latenzzeit	Gepuffert	Gültigkeitsbereich
Global Memory	off-chip	400-600 Takte	nein	Grid
Texture Memory	off-chip	400-600 Takte	ja	Grid
Constant Memory	off-chip	400-600 Takte	ja	Grid
Local Memory	off-chip	400-600 Takte	nein	Thread
Shared Memory	on-chip	1 Takt	nein	Block
Register	on-chip	1 Takt	nein	Thread

Tabelle 2.1: Speicherressourcen im Kernel-Kontext

Multiprozessoren und deren Ressourcen zu erreichen. So kann er bis zu acht Blöcke pro Multiprozessor steuern, solange die von den Blöcken benötigte Ressourcen kleiner als die vorhandenen Ressourcen eines Multiprozessors sind. Darüber hinaus regelt er den Ablauf von bis zu 768 Threads (24 Warps) pro Multiprozessor. Dadurch können die Latenzzeiten bei den Speicherzugriffen versteckt werden.

Abbildung 2.6 zeigt verschiedene Arten von Speicherzugriffen. So dauert ein sequenzieller Zugriff auf einen Speicher in einem Thread minimal $n * 400$ Takte und maximal $n * 600$ Takte. Kann der Scheduler die aktiven Threads, die auf ihre Werte warten, zur Seite legen und weitere Threads aktiv schalten, dann beträgt die parallel Zugriffszeit auf einen Speicherbereich bei n Threads $400 + (n - 1)$ Takte bzw. $600 + (n - 1)$ Takte.

4 sequentielle Lesezugriffe in einem Thread
Minimale Zugriffszeit: 4*400 Takte = 1600 Takte
Maximale Zugriffszeit: 4*600 Takte = 2400 Takte



4 Lesezugriffe in 4 Threads
Minimale Zugriffszeit: 400+1+1+1 Takte = 403 Takte
Maximale Zugriffszeit: 600+1+1+1 Takte = 603 Takte

Abbildung 2.6: Sequentielle und parallele Zugriffszeiten in CUDA

2.2 Image Mosaicing

Wie schon zuvor Erwähnt beruht der Kern des ImageJ-Plugins auf dem Optical Flow-Verfahren nach Kourogi[14]. Um ein grundlegendes Verständnis zu dem Verfahren zu bekommen, soll zunächst darauf eingegangen werden.

Kourogi benutzt zur Berechnung des globalen Bewegungsfeldes zwischen zwei konsekutiven Bildern so genannte *Pseudo Motion* und *Compensated Motion*. Der Ausgangspunkt seiner Überlegungen stellt die Gleichung des optischen Flusses dar:

$$I(x+u, y+v, t) - I(x, y, t-1) = 0 \tag{2.1}$$

So ist die Intensität[15] I des Bildpunktes x, y zum Zeitpunkt $t-1$ gleich der Intensität des Bildpunktes $x+u, y+v$ zum Zeitpunkt t. (u, v) bilden dabei einen Verschiebungsvektor. Er ist nicht bekannt und kann durch die Gleichung alleine nicht bestimmt werden. Deshalb schlägt Kourogi *Pseudo Motion* vor. So kann eine grobe Schätzungen des optischen Flusses an jedem Bildpunkt unternommen werden. Bei der Berechnung der Gleichung wird entweder der v- oder u-Term durch eine Null ersetzt. Somit ergeben sich folgende zwei Gleichungen zur Berechnung der Vektoren:

$$u_p = \frac{-I_t}{I_x} \tag{2.2}$$

$$v_p = \frac{-I_t}{I_y} \tag{2.3}$$

Um die horizontale (I_x), vertikale (I_y) und zeitliche (I_t) partielle Ableitungen der Helligkeit eines Bildpunktes zu bestimmen, verwendet Kourogi folgende Formeln:

$$I_y = \frac{I_r(x, y+\delta y) - I_r(x, y-\delta y)}{2} \tag{2.4}$$

$$I_x = \frac{I_r(x+\delta x, y) - I_r(x-\delta x, y)}{2} \tag{2.5}$$

$$I_t = I_c(x, y) - I_r(x, y) \tag{2.6}$$

I_r und I_c stellen dabei jeweils die Intensität des Referenz- bzw. des aktuellen Bildpunktes an der Stelle (x,y) dar.

[14] [Kourogi99]
[15] Auch als Lichtstärke oder Helligkeitswert bezeichnet

Nicht alle *Pseudo Motion*-Vektoren (u_p, v_p) sind für weiteren Berechnungen geeignet. Deshalb werden sie einem Test unterzogen, der einen Schwellenwert T zu der errechneten Intensitätsdifferenz eines Pixels des aktuellen Bildes zum Pixel des Referenzbildes gegenüberstellt:

$$|I_c(x + u_p, y + v_p) - I_r(x, y)| < T \tag{2.7}$$

Der Schwellenwert ist so zu wählen, dass der wahre *Pseudo Motion*-Vektor den Test besteht. Laut Kourogi erwies sich der empirisch ermittelte Schwellenwert 5 als ein guter Kompromiss zwischen der Echtzeitlauffähigkeit des Verfahrens und einer stabilen Schätzung der Vektoren. Alle *Pseudo Motion*-Vektoren, die diesen Test passieren werden zur weiteren Verarbeitung verwendet.

Mit Hilfe von *Psueudo Motion*-Vektoren lassen sich nur geringe Translationen von etwa einem Pixel schätzen. Bei größerer Translation wird das Ergebnis ungenau[16]. Deshalb schlägt Kourogi *Compensated Motion* als Lösung vor. Hierbei wird angenommen, dass das Verschiebungsfeld aus der Klasse der affinen Transformationen stammt, sodass eine globale *Compensated Motion* aus der *Pseudo Motion* geschätzt werden kann:

$$\begin{pmatrix} u_c \\ v_c \end{pmatrix} = \begin{pmatrix} a_1 x + a_2 y + a_3 \\ a_4 x + a_5 y + a_6 \end{pmatrix} \tag{2.8}$$

Es kann nun die geschätzte Bewegung zum Zeitpunkt t kompensiert werden. Die Gleichung für die partielle Ableitung nach der Zeit wird entsprechend verändert:

$$I_t^{(c)} = I_r(x + u_c, y + v_c) - I_c(x, y) \tag{2.9}$$

Die modifizierten *Pseudo Motion*-Vektoren berechnen sich als:

$$u_p = \frac{-I_t^{(c)}}{I_x} + u_c \tag{2.10}$$

$$v_p = \frac{-I_t^{(c)}}{I_y} + v_c \tag{2.11}$$

Die neu berechneten Vektoren u_p und v_p müssen wiederum nach Gleichung (2.7) getestet werden. Anschließend erfolgt eine Iteration der Berechnungen von u_c, v_c und u_p, v_p bis

[16] Vgl. [Konen06]

eine bestimmte Anzahl an Durchläufen erreicht wurde oder der durchschnittliche Fehler des Pixel-zu-Pixel-Vergleichs unter eine festgelegte Schwelle gesunken ist. Dies führt zu einer verbesserten Schätzung der Motion-Vektoren.

2.3 Kreuzkorrelationskoeffizient

Wann sind zwei Bilder gleich und wie kann man ihre Ähnlichkeit messen?

Das Vergleichen von Bildern ist kein einfaches Problem. Einem menschlichen Betrachter kommen zwei Bilder, welche sich voneinander durch eine geringe erhöhte Helligkeit oder eine geringe Rotation eines Bildes unterscheiden, gleich vor. Die numerische Differenz der beiden Bilder bzw. deren Teilausschnitte ist jedoch sehr verschieden. So geht es beim *Template Matching* um die Suche eines Bildmusters (Template) aus dem ersten Bild in dem zweiten Bild. Dabei wird das Template über das Bild verschoben und die Differenz gegenüber dem darunter liegenden Teilbild gemessen. Der Vorgang wird solange durchgeführt bis das Template mit dem Teilbild übereinstimmt bzw. dem ausreichend ähnlich ist. Die Differenz bezeichnet einen Intensitätsabstand zwischen zweidimensionalen Bildfunktionen.

Es gibt verschiedene gebräuchliche Definitionen zur Bestimmung der Differenz zwischen zweidimensionalen Bildfunktionen. Vollständigkeitshalber sollen hier Definitionen wie *Summe der Differenzbeträge*, *Maximaler Differenzbetrag*, *Summe der quadratischen Abstände*[17], sowie *Kreuzkorrelation* erwähnt werden[18]. Meist robustere Definitionen gegenüber globalen Intensitätsänderungen zwischen zwei Bildern stellen die *normalisierte Kreuzkorrelation* und der *Kreuzkorrelationskoeffizient* dar.

Die normalisierte Kreuzkorrelation kompensiert lokale Intensitätsänderungen im Gesamtbild, indem die Gesamtenergie im aktuellen Bildausschnitt berücksichtigt wird. Erhöht sich die Gesamthelligkeit des Bildes, in dem die Suche des Templates stattfindet, so ändert sich das Ergebnis der normalisierten Kreuzkorrelation dramatisch.

Der Kreuzkorrelationskoeffizient vermeidet dieses Problem, indem nicht nur die Differenz der ursprünglichen Funktionswerte miteinander verglichen werden, sondern die Differenz in Bezug auf die lokale Durchschnittswerte im Template einerseits und des zugehörigen Bildausschnitts im zu suchenden Bild andererseits[19].

[17] Auch bekannt als N-dimmensionaler euklidischer Abstand.
[18] Mehr Informationen sind [BB05], Kapitel 17, Seite 427 zu entnehmen.
[19] Vgl. [BB05], Kapitel 17, Seite 429.

Nach folgender Gleichung errechnet sich ein Kreuzkorrelationskoeffizient:

$$C(r,s) = \frac{\sum_{(i,j)\in R}(I(r+i,s+j) - I_d(r,s)) * (R(i,j) - R_d)}{\sqrt{\sum_{(i,j)\in R}(I(r+i,s+j) - I_d(r,s))^2} * \sqrt{\sum_{(i,j)\in R}(R(i,j) - R_d)^2}} \tag{2.12}$$

r und s stellen dabei einen Offset gegenüber den Template-Koordinaten dar und die Durchschnittswerte $I_d(r,s)$ und R_d werden wie folgt berechnet:

$$I_d(r,s) = \frac{1}{K} \sum_{(i,j)\in R} I(r+i,s+j) \tag{2.13}$$

$$R_d = \frac{1}{K} \sum_{(i,j)\in R} R(i,j) \tag{2.14}$$

K stellt die Anzahl der Elemente (Pixel) im Template R dar.

Die Ergebnisse der Gleichung 2.12 liegen im Intervall von -1 bis +1. Ein Wert $C(r,s) = +1$ zeigt dabei eine maximale Übereinstimmung und $C(r,s) = -1$ die maximale Abweichung zwischen dem Template R und dem aktuellen Bildausschnitt I an. Damit wird ein standardisiertes Maß für den Grad der Übereinstimmung geliefert, der direkt für die Entscheidung über die Akzeptanz der entsprechenden Position verwendet werden kann.

2.4 Logarithmic-Search

Logarithmic-Search stellt eine effiziente Suchmethode zur Bestimmung eines Bewegungsvektors (Motion Vector), der die optimale Translation eines bestimmten Templates im aktuellen Bild ausdrückt, dar.

Im Gegensatz zu einem Vollsuche-Algorithmus (Fullsearch), bei dem ein Template sukzessive im ganzen Bild gesucht wird, stellt Logarithmic-Search einen Schnellsuche-Algorithmus dar. Ein reduziertes Suchmusters ermöglicht dabei eine schnelle Bewegungsabschätzung (Motion Estimation) und den Fund eines Bewegungsvektors zu einem bestimmten Template.

Die Gesamtanzahl an Auswertungen N des Logarithmic-Search wird bestimmt durch die Anzahl an Iterationen i und der Entscheidungsmöglichkeiten S (Suchmuster):

$$N = i * S \tag{2.15}$$

Die Pixelanzahl p im Suchbereich ist:

$$p = S^i \tag{2.16}$$

Kombiniert man die Gleichungen 2.15 und 2.17, so ergibt sich die Komplexität des Logorithmic-Search, die proportional zum Logarithmus der Pixelanzahl und von einem Komplexitätsfaktor C_S abhängig ist[20]:

$$N = S * log_S(p) = \frac{S}{ln(S)} * ln(p) = C_S * ln(p) \tag{2.17}$$

Folgende fünf Schritte beschreiben die Funktionsweise des Logarithmic-Search:

Schritt 1

Festlegung des Startpunkts für die Suche und der 8 Punkte um den Startpunkt in dem Bild, in dem das Template gesucht wird. Für den Startpunkt werden meist die Koordinaten des Templates aus dem Ursprungsbild verwendet. Im ImageJ-Plugin wird jedoch der Startpunkt der Suche anhand einer initialen Schätzung durch die zuvor erläuterte Methode nach Kourogi festgelegt. Die 8 Punkte um den Startpunkt werden nach einem bestimmten horizontalen und vertikalen Offset angeordnet. Der Offset muss eine 2er Potenz aufweisen und größer als 1 sein. Die 9 Punkte stellen ein Suchmuster dar.

Schritt 2

Ermittlung des Grads der Übereinstimmung zwischen dem Template und dem Bildausschnitt im aktuellen Bild an den 9 Punkten des Suchmusters mit Hilfe eines bestimmten Maßes[21].

Schritt 3

Wenn der Punkt mit dem größten Grad der Übereinstimmung der Mittelpunkt des Suchmusters ist, dann geht es beim Schritt 4 weiter. Ansonsten wird der Punkt mit dem größten Grad der Übereinstimmung als neuer Mittelpunkt des Suchmusters festgelegt. Fortgefahren wird beim Schritt 2.

[20] Vgl. [Lundmark01], Seite 1.
[21] Ein Maß stellt z.B. der Kreuzkorrelationskoeffizient (Abschnitt 2.3) dar.

Schritt 4

Wenn der Offset = 1 ist, geht es beim Schritt 5 weiter.
Wenn der Offset > 1 ist, wird dieser durch 2 geteilt und die 8 Punkte des Suchmusters werden um den Punkt mit dem größten Grad der Übereinstimmung neu geordnet. Anschließend wird beim Schritt 2 fortgefahren.

Schritt 5

Der Bewegungsvektor wird auf den Punkt mit dem Grad der höchsten Übereinstimmung gesetzt.

Abbildung 2.7 zeigt ein Beispiel des Logarithmic-Search. Der Offset beträgt beim Start 4 Pixel. In dem Fall, dass der Grad mit der höchsten Übereinstimmung in der Mitte des Suchmusters liegt wird der Offset halbiert. Im ersten Suchschritt liegt der Punkt mit der besten Übereinstimmung in (i+4, j), somit wird das Suchmuster um den Punkt gelegt.

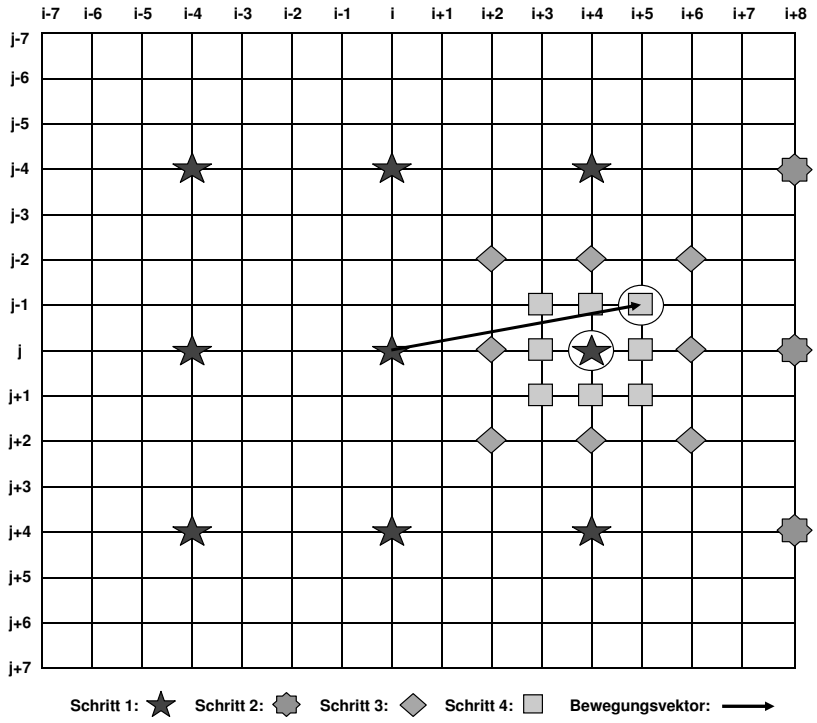


Abbildung 2.7: Beispiel einer Suche im Logarithmic-Search

Im zweiten und dritten Suchschritt wird der Offset halbiert, da der gesuchte Punkt in der Mitte des Suchmusters liegt. In der letzten Suchiteration liegt der Punkt in (i+5, j-1) und der Offset ist gleich 1, somit wurde das Ende der Suche erreicht. Der Bewegungsvektor wird auf (i+5, j-1) gesetzt.

2.5 Logarithmic-Search-Verfahren

Das neue Logarithmic-Search-Verfahren nach Wolfgang Konen, das im ImageJ-Plugin zum Einsatz kommt, basiert auf dem Logarithmic-Search und dem Kreuzkorrelationskoeffizient. Die Vorgehensweise des Algorithmus ist wie folgt:

1. Initiale Schätzung der Translation (u_c, v_c) innerhalb eines Bildpaars nach der Methode von Kourogi als Startpunkt für den Logarithmic Search.

2. Gleichmäßige Verteilung von $n * n$ Landmarken (xy-Koordinaten) auf dem Referenzbild, für die eine Entsprechung im aktuellen Bild gesucht wird.

3. Suche für jede Landmarke aus dem Referenzbild einen entsprechenden Bewegungsvektor (u_p, v_p) im aktuellen Bild mittels Logarithmic-Search, wobei der Kreuzkorellationskoeffizient C des Bewegungsvektors nicht kleiner als eine festgelegte Schranke (cthresh) sein darf:
$$C \geq cthresh \tag{2.18}$$

4. Bestehen weniger Kreuzkorrelationenskoeffizienten den cthresh-Test als eine festgelegte %-Anzahl (pGuar), so werden Bewegungsvetoren, die den Test nicht bestanden haben, mit den größten Kreuzkorrelationenskoeffizienten hinzugenommen.

5. Berechnung der globalen Affinen-Translation im Bildpaar, dadurch werden die Bewegungsvektoren (u_c, v_c) neu geschätzt.

6. Mit der euklidischen Distanz werden Bewegugsvektoren bestimmter Landmarken verworfen, deren distanz größer als eine Schranke (uthresh) ist:
$$\sqrt{(u_p - u_c)^2 + (v_p - v_c)^2} > uthresh \tag{2.19}$$

7. Wenn wiederum zu wenige Landmarken den uthresh-Test bestehen als eine festgelegte %-Anzahl (pGuar), so werden Landmarken mit der kleinsten euklidischen Distanz für weitere Berechnungen hinzugenommen.

8. Berechnung der globalen Affinen-Translation in dem Bildpaar, sowie der Bewegungsvektoren (u_c, v_c).

Die Tests *cthresh* und *uthresh* verhindern, dass bestimmte Ausreißer unter den Bewegungsvektoren der Landmarken mit in die Berechnung der globalen Translation einfließen. Eine gewisse %-Anzahl an Bewegungsvektoren wird dennoch für die Berechnung benötigt.

Kapitel 3

Optimierung des Logarithmic-Search-Verfahrens

Das Kapitel stellt Optimierungen des Logarithmic-Search-Verfahrens hinsichtlich der Performance sowie der Qualität dar.

Im ersten Abschnitt werden Performanceoptimierungen auf der CPU vorgestellt. Anschließend erfolgt eine Untersuchung der Parameter des Verfahrens, um die Qualität des zu erzeugenden Panoramabildes zu verbessern.

3.1 Methoden der Performancemessung

Die Messung der aktuellen Performance erfolgte zum einen mit einem Eclipse-Plugin *TPTP-Profiler* und zum anderen mit der Java-Methode *System.nanoTime*.

Ein *TPTP-Profiler* ermöglicht Messungen im gesamten Java-Programm durchzuführen. Als Ergebnis liefert er ein so genanntes *Call Tree*, das Ausführungszeiten der jeweiligen Methoden eines Programms prozentual und zeitlich in Bezug zur gesamten Ausführungszeit darstellt. Allerdings handelt es sich bei den zeitlichen Messungen nicht um die tatsächliche Ausführungszeit, da der *TPTP-Profiler* selbst Rechenleistung benötigt. Trotz der groben Übersicht, lassen sich rechenintensive Programmteile herausfinden, dessen Performance mit der Methode *System.nanoTime* genauer bestimmt werden kann.

Die Methode *System.nanoTime* liefert bei ihrer Ausführung die vergangene Zeit in Nanosekunden seit dem 01.01.1970 zurück. Wird die Methode zu zwei Zeitpunkten ausgeführt, so lässt sich die vergangene Zeit zwischen den Ausführungen bestimmen. Die Zeitspanne ergibt sich aus der Subtraktion der ersten von der zweiten Zeit.

3.2 Performance des Algorithmus

Anhand der Messung[1] der Ausführungszeit des ImageJ-Plugins durch den TPTP-Profiler, zusehen in Abbildung 3.1, lassen sich folgende rechenintensive Programmteile herausfiltern.

Call Tree

Thread Name	<Percent Per Thread	Cumulative Time (seconds)	Min Time	Avg Time	Max Time	Calls
AWT-EventQueue-0	100,00%	124,189399				
actionPerformed(java.awt.event.Actior	84,32%	104,713274	2,551857	52,356637	102,161417	2
process(Var, boolean, boolean, boo	82,22%	102,112033	102,112033	102,112033	102,112033	1
motion_log(ij.process.ImagePro	55,88%	69,401178	1,119013	1,334638	1,669343	52
search(ij.process.ImagePro	49,29%	61,212074	0,004678	0,023707	0,153974	2582
ls_cross(int[], int, int, ij.	48,36%	60,055372	0,001514	0,006262	0,150680	9590
crossCorrelation(ij.p	44,90%	55,761617	0,000157	0,000646	0,139405	86310
xMean(ij.proces	17,29%	21,476401	0,000061	0,000249	0,129846	86310
getPixels() java	7,72%	9,590789	0,000014	0,000056	0,138379	172620
getWidth() int	3,98%	4,937964	0,000014	0,000057	0,073815	86310
xMean(ij.process.Image	0,46%	0,577409	0,000062	0,000224	0,096915	2582
compareTo(java.lang.Objec	2,87%	3,564084	0,000037	0,000194	0,089788	18408
motion(ij.process.ImagePro	1,80%	2,231773	0,014322	0,042919	0,189115	52
lsAffine(int, double[][], doul	0,73%	0,904738	0,001768	0,008699	0,060208	104
Landmark(double, int, int, ir	0,20%	0,245718	0,000014	0,000060	0,036044	4124
meanInMask(double[][], dou	0,05%	0,062846	0,000462	0,000604	0,010026	104
arrTimes(double[][], double[	0,02%	0,023077	0,000015	0,000222	0,021166	104
clone() java.lang.Object	0,00%	0,002818	0,000046	0,000054	0,000122	52
getPixels() java.lang.Object	0,00%	0,001595	0,000014	0,000015	0,000093	104
sum(double[][]) double	0,00%	0,001522	0,000014	0,000015	0,000017	104
getWidth() int	0,00%	0,000986	0,000014	0,000019	0,000248	52
getHeight() int	0,00%	0,000790	0,000014	0,000015	0,000028	52
PanoAffine(ij.process.ImagePro	11,43%	14,189728	14,189728	14,189728	14,189728	1
getWidth() int	4,52%	5,610018	0,000014	0,000054	0,059128	103970
getMasks(ij.process.ImageProce	2,38%	2,949813	2,949813	2,949813	2,949813	1
openImage(java.lang.String) ij.	2,09%	2,598932	0,014464	0,046410	0,246360	56

Abbildung 3.1: Messergebnis des TPTP-Profilers für das ImageJ-Plugin zum Image Mosaicing

Die Methode *motion_log*, die das Logarithmic-Search-Verfahren und die Berechnung des Kreuzkorrelationskoeffizienten in sich beinhaltet, benötigt 55,88 % von der gesamten vergangenen Zeit des Threads der Anwendung.

Die 55,88 % teilen sich auf die beiden Bestandteile wie folgt. Die Berechnung der Kreuzkorrelationskoeffizienten erfolgt nach der Gleichung 2.12 und braucht 44,9 % per Thread. Der restliche Teil von 10,98 % verteilt sich auf das Logarithmic-Search-Verfahren (4,39 % per Thread) und andere Instruktionen in der Methode *motion_log* (5,61 % per Thread). Die rechenintensive Methode ist demnach *crossCorrelation*, die für das

[1] Als Basis für die Messung dienten Parametereinstellungen der Tabelle A.1, ein Computer mit Pentium M mit 1,4 GHz 2MB Cache und die Bildsequenz *Storz-parietal2* mit 52 Farbbildern.

Logarithmic-Search-Verfahren als Maß benötigt wird.

Tabelle 3.1 stellt Rechenzeiten der Methode *crossCorrelation* für die Berechnung eines Kreuzkorrelationskoeffizienten abhängig von der Seitenlänge eines Templates und für die Berechnung verwendeten Prozessor[2] dar. Gemessen wurden die Zeiten in Nanosekunden mit Hilfe der Methode *System.nanoTime*. Je nach Auflösung der zu verarbeitenden Bildfolge werden verschiedene Seitenlängen eines Templates verwendet. Bei Bildsequenzen mit einer Auflösung von 320x288 Pixel sind es Templates mit einer Seitenlänge zwischen 15 und 25 Pixel. Bei einer Bildauflösung von 720x576 Pixel benötigt man in der Regel einen zwischen 31x31 Pixel und 45x45 Pixel großen Template, um einen hohen Grad an Übereinstimmungen bei den Kreuzkorrelationskoeffizienten zu erzielen.

Template-Seitenlänge in Pixel	Rechenzeit in Nanosekunden
log.rsz	**Java Ver. 1.0**
15	6887
17	8430
19	10236
21	12108
23	14248
25	16483
27	18997
29	21791
31	24585
33	27834
35	30940
37	34362
39	38272
41	42194
43	46041
45	50015
47	54524
49	59183
51	63834

Tabelle 3.1: Berechnungszeiten für einen Kreuzkorrelationskoeffizient in Abhängigkeit der Seitenlänge eines Templates im unoptimierten Zustand

3.2.1 Methoden im unoptimierten Zustand

Ein geeignetes Maß für die Ähnlichkeit zwischen zwei Bildausschnitten stellt der Kreuzkorrelationskoeffizient dar. Er wird in der Methode *crossCorrelation* nach der Gleichung 2.12 berechnet. Dabei bekommt die Methode als Input ein konsekutives Grauwertbildpaar, bestehend aus einem Referenzbild und einem aktuellen Bild, sowie bestimmte

[2] Pentium M mit 1,4 GHz

Landmarken in Form von xy-Koordinaten. Die Koordinaten bestimmen dabei die Mitte der Teilbildausschnitte in den beiden Bildern.

Listing 3.1 zeigt einen Codeausschnitt aus der Methode *crossCorrelation*. Hierbei werden zunächst Referenzen auf die eindimensionalen Pixel-Arrays des Bildpaares aus den Bildprozessoren geholt (Zeile 2 und 3). Die Variable *tpixels* verweist demnach auf alle Pixel des Referezbildes und *ipixels* auf alle Pixel des aktuellen Bildes. Anschließend läuft die doppelte for-Schleife über alle Pixel eines bestimmten quadratischen Bildausschnitts im jeweiligen Bild. In Zeile 7 und 8 werden die Pixelzeilen der Bildausschnitte bestimmt und deren Grauwerte anschließend gelesen (Zeile 12 und 13). Um den Intensitätswert eines Pixels ohne Vorzeichen und im Bereich von 0 bis 255 zu erhalten, ist eine bitweise Maskierung mit $0xFF$ nötig. Von jedem Grauwert eines Pixels wird der lokale Durchschnittswert eines Bildausschnitts abgezogen. Die Berechnung der Durchschnittswerte *tMean*[3] und *iMean*[4] geschieht vorher in der Methode *xMean* nach den Gleichungen 2.13 und 2.14.

Der Kreuzkorrelationskoeffizient wird in der Zeile 20 berechnet und hat immer einen Wert zwischen -1 und 1.

3.2.2 Methoden im optimierten Zustand

Durch die Verwendung des schnellen eindimensionalen Pixel-Arrays, sowie der Berechnung von Pixelzeilen in der äußeren Schleife lässt sich der Originalcode der Methode *crossCorrelation* nicht weiter optimieren. Allerdings bringt eine Integration der Methode *xMean* in die Methode *crossCorrelation* eine Beschleunigung mit sich. Des weiteren erfolgt eine Optimierung durch eine Reduzierung der Aufrufe der Methode *crossCorrelation*. Die erwähnten Optimierungsschritte sollen folglich erläutert werden.

Optimierungsschritt 1

Betrachtet man die Gleichung 2.12 zur Berechnung des Kreuzkorrelationskoeffizienten genauer, so kann der Ausdruck, der die Varianz der Werte im Template R bestimmt, folgend geschrieben werden[5]:

[3] Einmal pro Template berechnet.
[4] Für jeden Bildausschnitt, mit dem ein Template verglichen wird, berechnet.
[5] Vgl. [BB05], Kapitel 17, Seite 430.

Listing 3.1: Code für die Berechnung eines Kreuzkorrelationskoeffizienten in Java

```
1  ...
2  byte [] tpixels = (byte[]) temp.getPixels();
3  byte [] ipixels = (byte[]) imag.getPixels();
4  int sizeXim = imag.getWidth();
5  int tRow, iRow;
6  for (int v=−offs;v<=offs; v++) {
7          tRow = (v+ty)∗sizeXim + tx − offs;
8          iRow = (v+iy)∗sizeXim + ix − offs;
9          for (int u=−offs; u<=offs; u++) {
10                 // tRow has value (v+ty)∗sizeXim+(u+tx)
11                 // iRow has value (v+iy)∗sizeXim+(u+ix)
12                 tc = ((0xFF & tpixels[tRow++])−tMean);
13                 ic = ((0xFF & ipixels[iRow++])−iMean);
14                 zSum+=tc∗ic;
15                 iCor+=ic∗ic;
16                 tCor+=tc∗tc;
17         }
18 }
19 ...
20 return (zSum/(Math.sqrt(tCor∗iCor)));
```

$$\sum_{(i,j)\in R} (R(i,j) - R_d)^2 = \sum_{(i,j)\in R} (R(i,j))^2 - K * (R_d)^2$$

dementsprechend ist

$$\sum_{(i,j)\in R} (I(r+i,s+j) - I_d(r,s)) = \sum_{(i,j)\in R} (I(r+i,s+j))^2 - K * (I_d(r,s))^2$$

Durch eine Ersetzung in Gleichung 2.12 ergibt sich eine effiziente Möglichkeit zur Berechnung der lokalen Kreuzkorrelationskoeffizienten:

$$C(r,s) =$$

$$\frac{\sum_{(i,j)\in R} (I(r+i,s+j) * R(i,j)) - K * I_d(r,s) * R_d}{\sqrt{\sum_{(i,j)\in R} (I(r+i,s+j))^2 \ - K * (I_d(r,s))^2} * \sqrt{\sum_{(i,j)\in R} (R(i,j))^2 \ - K * (R_d)^2}} \quad (3.1)$$

Die lokalen Durchschnittswerte des Bildausschnitts ($I_d(r,s)$) im aktuellen Bild und des Templates (R_d) im Referenzbild können wie folgt berechnet werden:

$$I_d(r,s) = \frac{1}{K} \sum_{(i,j)\in R} I(r+i, s+j) \tag{3.2}$$

$$R_d = \frac{1}{K} \sum_{(i,j)\in R} R(i,j) \tag{3.3}$$

Der Gesamte Ausdruck in Gleichung 3.1 lässt sich in einer einzigen Doppel-Schleife berechnen.

Der neue optimierte Code der Methode *crossCorrelation* abgebildet in Listing 3.2 beinhaltet 2 Schritte zur Berechnung eines Kreuzkorrelationskoeffizienten. Im ersten Schritt werden Zwischensummen gebildet (Zeile 16 bis 20). Im zweiten Schritt lässt sich dann der Kreuzkorrelationskoeffizient ausrechnen (Zeile 24 und 25). Dabei weicht das Ergebnis der optimierten Version gegenüber der original Version nach der zwölften Nachkommastelle ab. Es hängt damit zusammen, dass Double-Rechenoperationen nicht kommutativ sind.

Optimierungsschritt 2

Der optimierte Java Quellcode aus dem Listing 3.2 wurde mit Hilfe von JNI[6] in die native Programmiersprache C überführt.

Listing 3.3 zeigt einen C-Code zur Berechnung eines Kreuzkorrelationskoeffizienten. In Zeile 3 und 4 werden Zeiger auf eindimensionale Arrays des Bildpaares über JNI spezifische Methoden angelegt. Diese müssen explizit wieder gelöst werden (Zeile 20 und 21). In Zeilen 10 bis 18 werden, wie schon im Schritt 1 beschrieben, Pixelgrauwerte aus dem Template des Referenzbildes und dem Bildausschnitt des aktuellen Bildes für die Zwischensummenberechnungen geholt, wobei die Double-Cast-Operation in C angegeben werden muss. In Java ist die Angabe der Cast-Operation vom Byte ins Double an der Stelle nicht erforderlich. Zum Schluss erfolgt die Berechnung des Kreuzkorrelationskoeffizienten (Zeile 23 und 24).

[6] Mehr zu Java Native Interface im Abschnitt 4.1

Listing 3.2: Optimierter Code für die Berechnung des Kreuzkorrelationskoeffizienten in Java

```
1  double valT=0, valC=0, sumT =0, sumT2=0, sumC =0, sumC2=0, prodCT=0;
2  double pixelnumber = w*h;
3  byte [] tpixels = (byte[]) temp.getPixels();
4  byte [] ipixels = (byte[]) imag.getPixels();
5  int sizeXim = imag.getWidth();
6  int tRow, iRow;
7
8  for (int v=-offs; v<=offs; v++) {
9          tRow = (v+ty)*sizeXim + tx - offs;
10         iRow = (v+iy)*sizeXim + ix - offs;
11         for (int u=-offs; u<=offs; u++) {
12                 // tRow has value (v+ty)*sizeXim+(u+tx)
13                 // iRow has value (v+iy)*sizeXim+(u+ix)
14                 valT = (0xFF & tpixels[tRow++]);
15                 valC = (0xFF & ipixels[iRow++]);
16                 sumT += valT;
17                 sumT2 += valT*valT;
18                 sumC += valC;
19                 sumC2 += valC*valC;
20                 prodCT += valT*valC;
21         }
22 }
23
24 return (prodCT - sumC * sumT/pixelnumber)
25       /Math.sqrt((sumC2 - sumC*sumC/pixelnumber)*(sumT2 - sumT*sumT/pixelnumber));
```

Optimierungsschritt 3

Eine genaue Betrachtung des implementierten Logarithmic-Search-Verfahrens in den Methoden *search* und *ls_cross* ergab, dass dieser nicht effizient arbeitet. Die Uneffizienz besteht darin, die schon einmal ausgerechneten Kreuzkorrelationskoeffizienten nicht wiederzuverwenden.

Am Beispiel in der Abbildung 2.7 ist zu sehen, dass einige der im ersten Schritt berechneten Kreuzkorrelationskoeffizienten im zweiten ((i, j-4), (i+4, j-4), (i, j), (i+4, j), (i, j+4), (i+4, j+4)) im dritten ((i+4, j) und im vierten Schritt (i+4, j) wiederverwenden werden können. Dadurch verringert sich die Zeit für die Suche des Bewegungsvektors (i+5, j)).

Listing 3.4 zeigt einen Ausschnitt aus dem Java-Code zu dem optimierten Logarithmic-

Listing 3.3: Optimierter Code für die Berechnung des Kreuzkorrelationskoeffizienten in C

```
1  ...//Variablen Initialisierung
2  jboolean isCopy = JNI_FALSE;
3  jbyte *tpixels = (jbyte*) env->GetPrimitiveArrayCritical(JNI_t_bytepixels, &isCopy);
4  jbyte *cpixels = (jbyte*) env->GetPrimitiveArrayCritical(JNI_c_bytepixels, &isCopy);
5
6  for (int v=-JNI_offs; v<=JNI_offs; v++) {
7          tRow = (v+JNI_ty)*JNI_sizeXim + JNI_tx - JNI_offs;
8          iRow = (v+JNI_cy)*JNI_sizeXim + JNI_cx - JNI_offs;
9          for (int u=-JNI_offs; u<=JNI_offs; u++) {
10                 valT = (double)(0xFF & tpixels[tRow++]);
11                 valC = (double)(0xFF & cpixels[iRow++]);
12                 sumT += valT;
13                 sumT2 += valT*valT;
14                 sumC += valC;
15                 sumC2 += valC*valC;
16                 prodCT += valT*valC;
17         }
18 }
19
20 env->ReleasePrimitiveArrayCritical(JNI_t_bytepixels, tpixels, JNI_ABORT);
21 env->ReleasePrimitiveArrayCritical(JNI_c_bytepixels, cpixels, JNI_ABORT);
22
23 return (prodCT - sumC * sumT/pixelnumber)
24       /sqrt((sumC2 - sumC*sumC/pixelnumber)*(sumT2 - sumT*sumT/pixelnumber));
```

Search-Verfahren. In den Zeilen 2 bis 9 werden 8 Punkte aus dem Suchmuster um den Mittelpunkt gesetzt, wenn diese nicht außerhalb einer bestimmten Maske liegen.

In der for-Schleife in der Zeile 11 wurde im unoptimierten Code für jeden der 9 Punkte des Suchmusters die Methode *crossCorrelation* ausgeführt. Im optimierten Zustand findet dagegen die Berechnung des Kreuzkorrelationskoeffizienten für einen bestimmten Punkt nur dann statt, wenn dieser zuvor nicht ermittelt wurde. Dazu wurde die globale zweidimensionale Hilfsvariable *acceptCC* eingeführt, die für jedes Bildpaar mit 0 initialisiert ist.

Wird für einen Punkt, der durch eine xy-Koordinate bestimmt ist, ein Kreuzkorrelationskoeffizient berechnet (Zeile 13), so lässt sich der Wert in die Hilfsvariable an die xy-Stelle schreiben (Zeile 14). In den nachfolgenden Iterationen des Logarithmic-Search-Verfahrens wird der zuvor berechnete Wert für einen bestimmten Punkt wieder gelesen,

falls benötigt (Zeile 16).

Listing 3.4: Optimierter Code des Logarithmic-Search-Verfahrens mit einer Wiederverwendung von berechneten Kreuzkorrelationskoeffizienten

```
1  ...
2  if (maskn_c[p[0]−s][p[1] ]==1) { vec[1][0] = p[0]−s; vec[1][1] = p[1] ; }
3  if (maskn_c[p[0]−s][p[1]−s]==1) { vec[2][0] = p[0]−s; vec[2][1] = p[1]−s; }
4  if (maskn_c[p[0] ][p[1]−s]==1) { vec[3][0] = p[0] ; vec[3][1] = p[1]−s; }
5  if (maskn_c[p[0]+s][p[1]−s]==1) { vec[4][0] = p[0]+s; vec[4][1] = p[1]−s; }
6  if (maskn_c[p[0]+s][p[1] ]==1) { vec[5][0] = p[0]+s; vec[5][1] = p[1] ; }
7  if (maskn_c[p[0]+s][p[1]+s]==1) { vec[6][0] = p[0]+s; vec[6][1] = p[1]+s; }
8  if (maskn_c[p[0] ][p[1]+s]==1) { vec[7][0] = p[0] ; vec[7][1] = p[1]+s; }
9  if (maskn_c[p[0]−s][p[1]+s]==1) { vec[8][0] = p[0]−s; vec[8][1] = p[1]+s; }
10
11 for (int i=0; i<9; i++) {
12         if(Proc.acceptCC[vec[i][0]][vec[i][1]]==0){
13          cvec[i] = crossCorrelation(temp,t[1],t[0], imag,vec[i][1],vec[i][0], 2∗o+1, 2∗o+1,o);
14          Proc.acceptCC[vec[i][0]][vec[i][1]]=cvec[i];
15         }
16         else cvec[i]=Proc.acceptCC[vec[i][0]][vec[i][1]];
17 }
18 ...
```

3.2.3 Performance Ergebnisse

Optimierungsschritt 1 und 2

Die Tabelle 3.2 stellt Zeiten für die Berechnung der Kreuzkorrelationskoeffizienten für die zuvor dargelegten Optimierungsschritte zusammen. Die Messung fand auf einem Pentium M mit 1,4 GHz statt. Die Tabellenspalte *Java Ver. 1.0* enthält Rechenzeiten der ersten Version der Methode *crossCorrelation* ohne jegliche Optimierung. Die Spalte *Java Ver. 1.1* stellt die Rechenzeiten für den im Schritt eins optimierten Java-Code dar. Der zweite Optimierungsschritt bringt die in der Spalte *JNI C* aufgeführten Zeiten mit sich. Die Zeiten sind jeweils von der Seitenlänge eines Templates, die in Pixel angegeben ist, abhängig. Aufgrund der Abhängigkeit steigen die Berechnungszeiten quadratisch an.

Durch den ersten Optimierungsschritt konnte die Methode *crossCorrelation* um den Faktor zwischen 1,29 und 1,37 beschleunigt werden (siehe Abb. 3.2). Dabei liegt der mittlere Beschleunigungsfaktor bei 1,356.

Der Einsatz der Sprache *C* im zweiten Optimierungsschritt beschleunigte den Algo-

Template-Seitenlänge in Pixel	Rechenzeit in Nanosekunden			Beschleunigungsfaktor X	
log.rsz	**Java Ver. 1.0**	**Java Ver. 1.1**	**JNI C**	**Java Ver. 1.1 zu 1.0**	**JNI C zu Java Ver. 1.0**
15	6887	5308	4048	1.29748	1.70133
17	8430	6426	4511	1.31186	1.86877
19	10236	7543	5251	1.35702	1.94934
21	12108	8940	5711	1.35436	2.12012
23	14248	10616	6515	1.34213	2.18695
25	16483	12293	7219	1.34084	2.28328
27	18997	14107	8156	1.34664	2.32921
29	21791	16063	8892	1.35660	2.45063
31	24585	18159	10112	1.35387	2.43127
33	27834	20394	11287	1.36481	2.46602
35	30940	22629	12305	1.36727	2.51443
37	34362	25143	13499	1.36666	2.54552
39	38272	27796	14459	1.37689	2.64693
41	42194	30591	15480	1.37929	2.72571
43	46041	33524	17035	1.37337	2.70273
45	50015	36597	18140	1.36664	2.75717
47	54524	39670	19704	1.37444	2.76715
49	59183	43023	21056	1.37561	2.81074
51	63834	46514	22758	1.37236	2.80490

Tabelle 3.2: Berechnungszeiten der Methode *crossCorrelation* in unterschiedlichen Versionen auf einem Pentium M mit 1,4 GHz in Abhängigkeit der Template-Größe

rithmus gegenüber der unoptimierten Version um den Faktor 1,7 bis 2,81, je nach größe eines Templates. Der mittlere Beschleunigungsfaktor liegt hier bei 2,424.

Trotz eines kleinen Overhead, der durch das Schleusen der Daten über JNI entsteht, kann sich das Ergebnis der Sprache C gegenüber Java sehen lassen.

Ein etwas kleinerer Beschleunigugsunterschied erweist sich bei der Verwendung eines schnelleren Rechners (Athlon64 X2 2,5 GHz). Tabelle 3.3 und die Illustrierung dieser in Abbildung 3.3 zeigen eine kleinere Spreizung der Beschleunigung der optimierter Java-Version gegenüber der C-Version. Der optimierte Java Algorithmus erreicht einen Beschleunigungsfaktor zwischen 1,41 und 1,60, bei einem Mittelwert von ca 1,54. Die native Sprache C beschleunigt den Algorithmus um den Faktor 1,47 bis 2,22, wobei der Mittelwert sich bei 1,97 befindet. Im Bereich von 15 bis 17 Pixel pro Seitenlänge eines Templates ist die Methode in unterschiedlichen Sprachen gleich schnell. Bei größeren Templates ist C wiederum schneller als Java.

Optimierungsschritt 3

Tabelle 3.4 listet in den Spalten *ls_cross Ver. 1.0* und *ls_cross Ver. 1.1* eine durch-

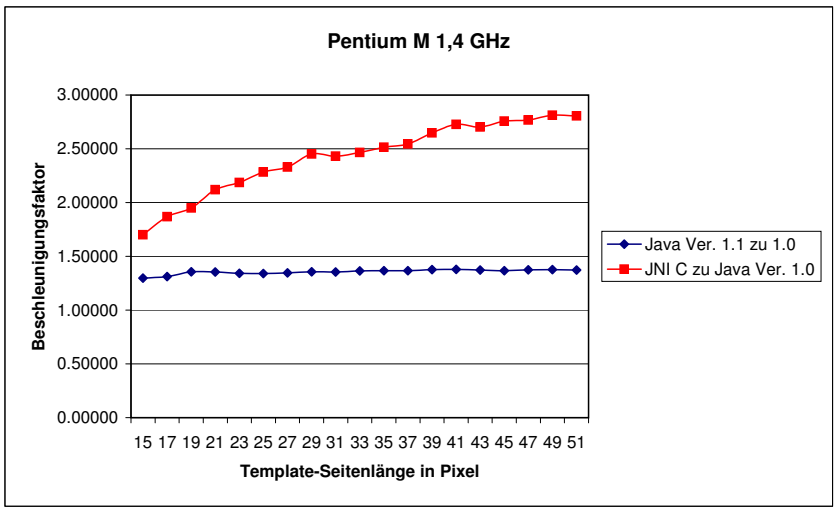


Abbildung 3.2: Beschleunigungsfaktoren optimierter Methode *crossCorrelation* in unterschiedlichen Versionen auf einem Pentium M mit 1,4 GHz in Abhängigkeit der Template-Größe

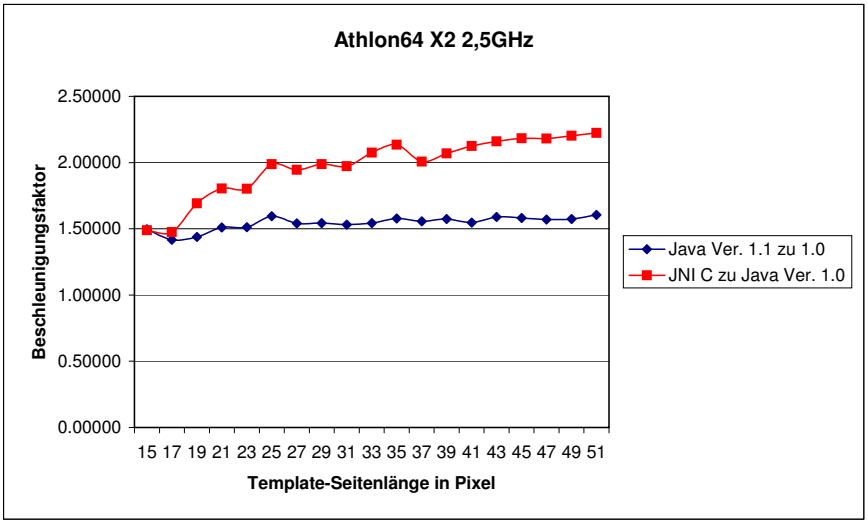


Abbildung 3.3: Beschleunigungsfaktoren optimierter Methode *crossCorrelation* in unterschiedlichen Versionen auf einem Athlon64 X2 2,5 GHz in Abhängigkeit der Template-Größe

Template-Seitenlänge in Pixel	Rechenzeit in Nanosekunden			Beschleunigungsfaktor X	
log.rsz	**Java Ver. 1.0**	**Java Ver. 1.1**	**JNI C**	**Java Ver. 1.1 zu 1.0**	**JNI C zu Java Ver. 1.0**
15	3007	2010	2019	1.49602	1.48935
17	3370	2379	2285	1.41656	1.47484
19	4433	3083	2620	1.43789	1.69198
21	5238	3468	2904	1.51038	1.80372
23	5910	3910	3280	1.51151	1.80183
25	7094	4448	3569	1.59487	1.98767
27	7944	5154	4086	1.54133	1.94420
29	8929	5785	4492	1.54347	1.98776
31	10110	6605	5127	1.53066	1.97191
33	11329	7347	5459	1.54199	2.07529
35	12672	8032	5935	1.57769	2.13513
37	13992	8985	6972	1.55726	2.00688
39	15384	9769	7439	1.57478	2.06802
41	16951	10963	7980	1.54620	2.12419
43	18705	11768	8664	1.58948	2.15893
45	20090	12708	9206	1.58089	2.18227
47	21765	13857	9978	1.57069	2.18130
49	23629	15017	10731	1.57348	2.20194
51	25737	16028	11572	1.60575	2.22408

Tabelle 3.3: Berechnungszeiten der Methode *crossCorrelation* in unterschiedlichen Versionen auf einem Athlon64 X2 2,5 GHz in Abhängigkeit der Template-Größe

schnittliche Anzahl der Kreuzkorrelationskoeffizienten-Berechnungen pro Bildpaar für die Testsequenz *Storz_parietal2*[7] auf. Die Anzahl der Berechnungen variiert dabei je nach Template-Größe und hängt lediglich von den Eigenschaften einer Testsequenz ab. Durch die Wiederverwendung von zuvor berechneten Kreuzkorrelationskoeffizienten in der optimierten Methode *ls_cross* ergibt sich ein mittlerer Beschleunigungsfaktor von 1.38. Wie der Abbildung 3.4 zu entnehmen, verhält sich die Beschleunigung bei halber PAL-Auflösung reziprok zu den Template-Größen.

Durchschnittliche Anzahl der Kreuzkorrelationskoeffizienten-Berechnungen pro Bildpaar für die Testsequenz *Versuch 1 klein*[8] zeigt die Tabelle 3.5. Bei höheren Auflösung ist die Wiederverwendung von zuvor errechneten Kreuzkorrelationskoeffizienten proportional zu den Temlategrößen, sodass die Beschleunigung sich auf einem Level hält (siehe Abb. 3.5). Der mittlere Beschleunigungsfaktor liegt hier bei ca 1.61.

[7] Auflösung: 360x288, Bildpaare: 51
[8] Auflösung: 720x576, Bildpaare: 9

Template-Seitenlänge in Pixel	Suchpunkte pro Bildpaar		Beschleunigungsfaktor
log.rsz	**ls_cross Ver. 1.0**	**ls_cross Ver. 1.1**	**ls_cross Ver. 1.1 zu 1.0**
15	1872	1276	1.46708
17	1770	1236	1.43204
19	1697	1204	1.40947
21	1682	1198	1.40401
23	1708	1221	1.39885
25	1722	1214	1.41845
27	1658	1183	1.40152
29	1698	1218	1.39409
31	1665	1203	1.38404
33	1641	1187	1.38248
35	1608	1175	1.36851
37	1635	1195	1.36820
39	1631	1193	1.36714
41	1576	1164	1.35395
43	1606	1182	1.35871
45	1603	1187	1.35046
47	1549	1162	1.33305
49	1564	1159	1.34944
51	1565	1161	1.34798

Tabelle 3.4: Durchschnittliche Anzahl der Kreuzkorrelationskoeffizienten-Berechnungen pro Bildpaar für die Testsequenz “Storz_parietal2“

Template-Seitenlänge in Pixel	Suchpunkte pro Bildpaar		Beschleunigungsfaktor
log.rsz	**ls_cross Ver. 1.0**	**ls_cross Ver. 1.1**	**ls_cross Ver. 1.1 zu 1.0**
15	2484	1537	1.61614
17	2436	1510	1.61325
19	2431	1502	1.61851
21	2379	1504	1.58178
23	2442	1511	1.61615
25	2471	1501	1.64624
27	2405	1434	1.67713
29	2332	1476	1.57995
31	2329	1445	1.61176
33	2268	1446	1.56846
35	2288	1482	1.54386
37	2296	1439	1.59555
39	2280	1406	1.62162
41	2366	1462	1.61833
43	2241	1395	1.60645
45	2150	1375	1.56364
47	2368	1442	1.64216
49	2212	1405	1.57438
51	2229	1375	1.62109

Tabelle 3.5: Durchschnittliche Anzahl der Kreuzkorrelationskoeffizienten-Berechnungen pro Bildpaar für die Testsequenz „Versuch 1 klein“

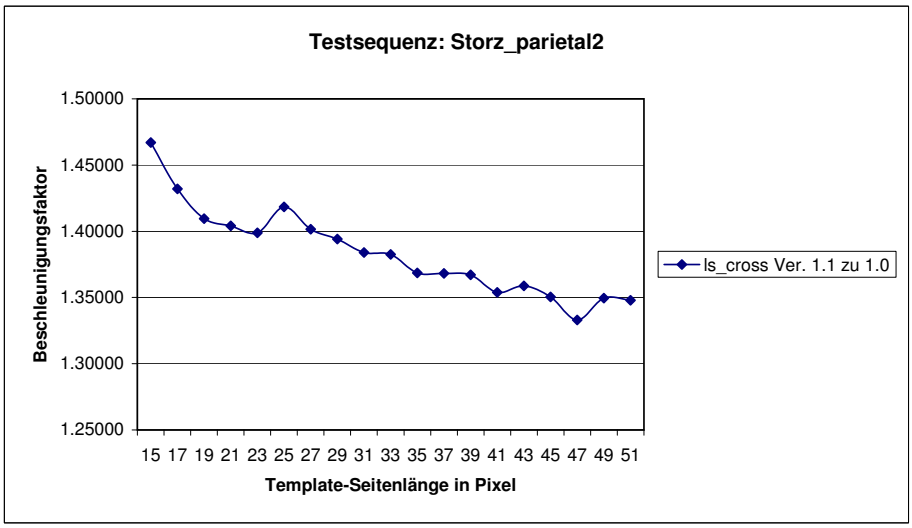


Abbildung 3.4: Beschleunigungsfaktoren optimierter Methode *ls_cross* in Abhängigkeit der Template-Größe für die Testsequenz "Storz_parietal2"

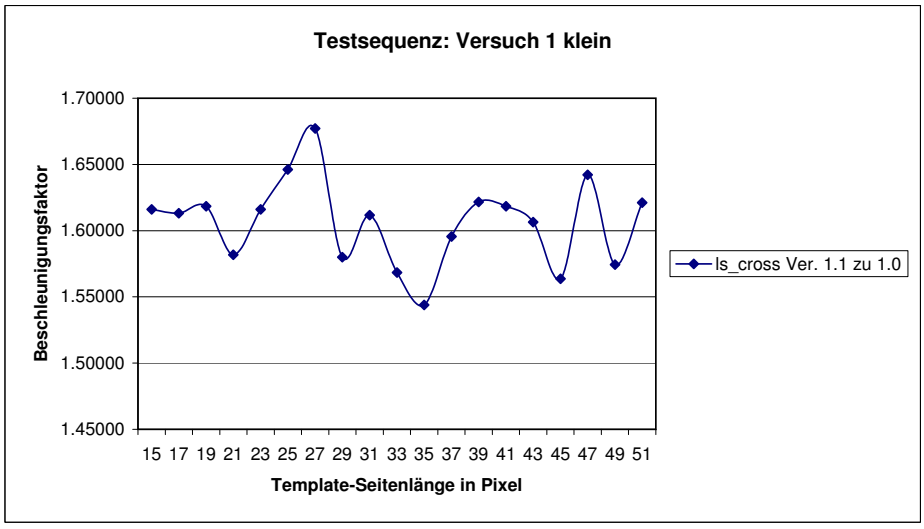


Abbildung 3.5: Beschleunigungsfaktoren optimierter Methode *ls_cross* in Abhängigkeit der Template-Größe für die Testsequenz „Versuch 1 klein"

3.3 Parameter des Algorithmus

Im ImageJ-Plugin implementierter Logarithmic-Search-Verfahren wird über zahlreiche Parameter gesteuert, die sich auf die Güte des Matches zwischen einem Referenzbild und einem aktuellen Bild auswirken.

Zu den kritischen Parametern gehören :

- **log.rsz** - stellt die Seitenlänge eines quadratischen Templates dar.
- **log.cthresh** - ist ein Grenzwert, der den minimalen zulässigen Wert bei einem Kreuzkorrelationskoeffizient eines Bewegungsvektors angibt. Das heißt, dass alle Bewegungsvektoren mit einem Kreuzkorrelationskoeffizient unter diesem Wert nicht in die Berechnung des Panoramabildes einfließen.
- **log.NPTS** - bestimmt die Anzahl der Templates in einem Referenzbild, die im aktuellen Bild mittels Logarithmic-Search gesucht werden. Diese Anzahl wird quadriert und über das ganze Referenzbild gleichmäßig verteilt.

Im Folgenden wird ein Vergleichsmaß vorgestellt, durch den die Parametrisierung des Logarithmic-Search-Verfahrens mit den zuvor genannten Parametern gemessen werden kann. Anschließend erfolgt die Auswertung der Messergebnisse.

3.3.1 Vergleichsmaß

Jannis Bloemendal untersuchte in seiner Diplomarbeit[9] schnelle Verfahren zur Objektregistrierung. Für einen Vergleich der registrierten Bilder verwendete er drei verschiedene Vergleichsmaße. Dabei stellte sich heraus, dass ein Vergleichsmaß auf Basis des Kreuzkorrelationskoeffizienten den visuellen Eindruck der Registrierung am Besten wiedergibt.

Bei dem Verfahren wird auf zwei Bilder eine gleichgroße Boundingbox gelegt, die in $n * n$ gleichgroße Templates aufgeteilt ist (siehe Abbildung 3.6). Anschließend wird für alle n^2 zueinander gehörende Templatepaare der Kreuzkorrelationskoeffizient berechnet. Abschließend bildet man einen Mittelwert aus allen Kreuzkorrelationskoeffizienten, dessen Wert zwischen 1 und -1 liegt. Ein guter Match zwischen zwei Bildern stellt die 1 dar. Ein Minuswert deutet dagegen auf einen miserablen Match hin. Nach [Bloem09] liefert eine Boundingbox mit $5*5$ Templates die optimale Konfiguration für das Vergleichsmaß.

[9] [Bloem09]

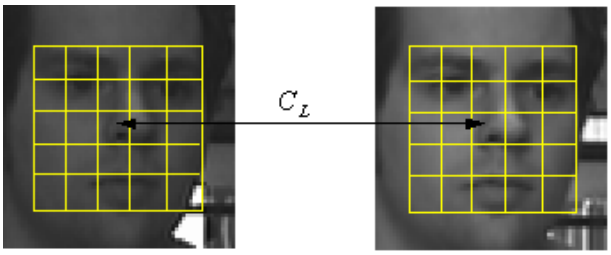


Abbildung 3.6: Vergleichsmaß auf Basis von Kreuzkorrelationskoeffizienten [Bloem09]

Um eine Bewertung der Güte des Panoramabildes zu ermöglichen, wurde das Vergleichsmaß in das ImageJ-Plugin implementiert.

3.3.2 Qualitative Ergebnisse

Das Logarithmic-Search-Verfahren wurde durch eine systematische Variation der zuvor vorgestellten Parameter auf bestmögliches Matching hin untersucht. Falls nicht anders erwähnt, wurden die restlichen Parameter des ImageJ-Plugins, wie im Anhang A in der Tabelle A.1 dargestellt, gesetzt. Die Untersuchung fand an zwei Testsequenzen statt: *Storz_parietal2*[10] und *Versuch 1 klein*[11].

Parameter log.rsz und log.cthresh

Tabelle 3.7 zeigt die Ergebnisse des zuvor vorgestellten Vergleichsmaßes für die Testsequenz *Storz_parietal2*. Dabei wurden die Parameter *log.rsz* und *log.cthresh* systematisch variiert. Die Güte der Panoramabilder stimmte nicht immer mit dem visuellen Eindruck überein, sodass die Ausreißer in den Spalten *log.cthresh=0.70* und *log.cthresh=0.75* farblich gekennzeichnet wurden. Abbildung 3.8 illustriert die Güte der Panoramalbilder in Abhängigkeit der Seitenlänge eines Templates. Daraus ist ersichtlich, dass die Güte bis zu einer Template-Seitenlänge von 25 Pixel ansteigt und danach mit zunehmender Seitenlänge abnimmt, wobei die Spitzenwerte in der Tabelle 3.7 als farbliche Zellen markiert sind. Die Anzahl der Zellenmarkierungen in einer Spalte deutet auf bestmögliche Konfiguration des Parameters *log.cthresh* hin. Eine hohe Güte des Panoramabildes über verschiedene Template-Seitenlängen gibt es demnach bei einem Wert von 0,85 des Parameters *log.cthresh*.

[10] Auflösung: 360x288, Bilder: 52
[11] Auflösung: 720x576, Bilder: 10

log.rsz	log.cthresh = 0.70	log.cthresh = 0.75	log.cthresh = 0.80	log.cthresh = 0.85	log.cthresh = 0.90	Mittelwert
15	0.749765998	0.747606506	0.747318393	0.743591685	0.743034044	0.74464804
17	0.749214433	0.746490859	0.741431553	0.733854201	0.74444076	0.739908838
19	0.755328867	0.74942661	0.747059979	0.745162684	0.753428716	0.74855046
21	0.756705935	0.753930255	0.755303457	0.753364299	0.758042141	0.755569966
23	0.755717335	0.754961229	0.751015172	0.753005415	0.751388291	0.751802959
25	0.761679976	0.763073623	0.76187423	0.762569043	0.761378521	0.761940598
27	0.752858578	0.753303959	0.753969018	0.754175259	0.750783881	0.752976053
29	0.75919053	0.758747939	0.760147544	0.758239275	0.755262969	0.757883263
31	0.746414813	0.74730869	0.74831652	0.746645209	0.74735289	0.747438206
33	0.745254195	0.746768327	0.757756764	0.755142406	0.754505085	0.755801418
35	0.752229575	0.752737095	0.751634355	0.752062278	0.750724146	0.751473593
37	0.752516671	0.75231723	0.752485682	0.75344546	0.751061329	0.752330824
39	0.739679917	0.740831841	0.741660935	0.741848852	0.740764048	0.741424612
41	0.74826733	0.749505267	0.750228897	0.750868384	0.752748494	0.751281925
43	0.745918268	0.750178679	0.751573936	0.751767562	0.749749499	0.751030332
45	0.740079447	0.741263977	0.744986257	0.746322941	0.747660607	0.746323268
47	0.745657324	0.745623353	0.747360631	0.749032253	0.751882971	0.749425285

Tabelle 3.6: Güte des Panoramabildes für die Testsequenz “Storz_parietal2“ abhängig von den Parametern *log.rsz* und *log.cthresh*

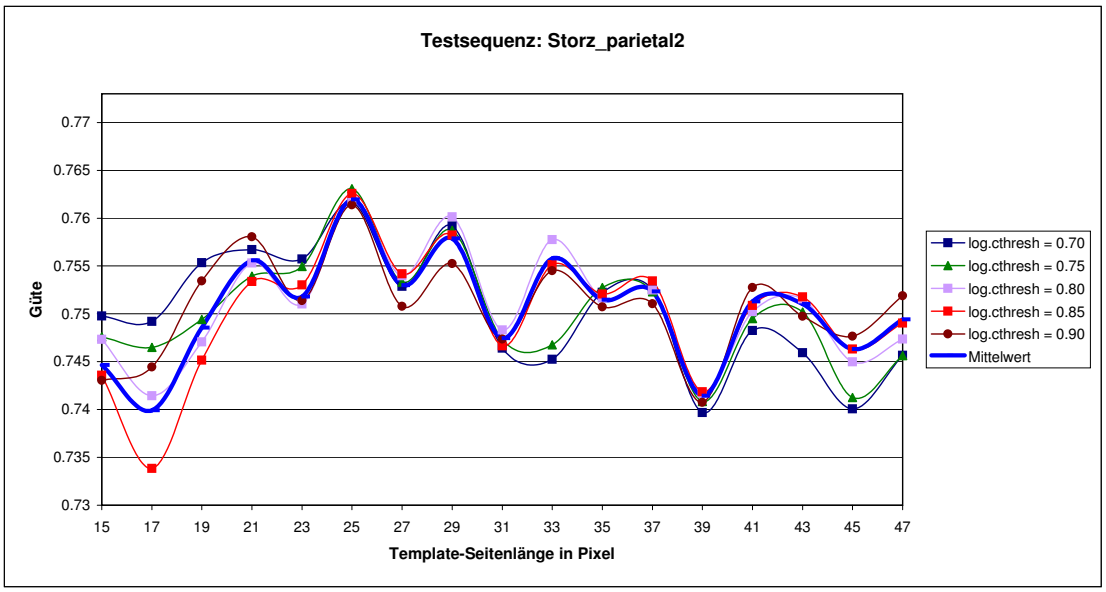


Abbildung 3.7: Güte des Panoramabildes für die Testsequenz “Storz_parietal2“ abhängig von den Parametern *log.rsz* und *log.cthresh*

Die Variation der Güte des Panoramabildes für eine großformatige Testsequenz *Versuch 1 klein* in Abhängigkeit der beiden Parameter *log.rsz* und *log.cthresh* zeigt die Tabelle 3.7. Die visuellen Ausreißer bei den *log.cthresh*-Werten 0,7 und 0.75 sind wiederum farblich gekennzeichnet und werden bei der Auswertung nicht berücksichtigt. Die farblichen Zellen markieren die höchste Güte eines Panoramabildes in Abhängigkeit der Seitenlänge eines Templates. Abbildung 3.8 veranschaulicht die Zusammenhänge der beiden Parameter. So ergeben sich im Template-Seitenlängenbereich von 31 bis 33 Pixel die bestmöglichen Panoramabilder, rechnerisch aber auch vom visuellen Eindruck. Mit zunehmender Größe der Templates wird die Qualität des Matches abgeschwächt. Der *log.cthresh*-Parameterwert von 0,85 gibt die bestmögliche Güte eines Panoramabildes her, sodass der Wert für klein- und großformatige Sequenzen geeignet ist.

log.rsz	log.cthresh = 0.70	log.cthresh = 0.75	log.cthresh = 0.80	log.cthresh = 0.85	log.cthresh = 0.90	Mittelwert
25	0.603331649	0.597554521	0.592629971	0.612178417	0.587107112	0.598560334
27	0.617235359	0.617128035	0.626370949	0.562080024	0.58166059	0.600894992
29	0.58685872	0.576575051	0.57991635	0.602681	0.562117588	0.581629742
31	0.629072275	0.620054454	0.636684739	0.639906687	0.623894248	0.62992248
33	0.606133273	0.621340002	0.615354685	0.639943945	0.615099835	0.619574348
35	0.601179831	0.596050538	0.610218693	0.626286515	0.615026214	0.609752358
37	0.590563133	0.585908964	0.590027052	0.61307358	0.608138376	0.597542221
39	0.607583549	0.609131051	0.609219542	0.625193897	0.62186503	0.614598614
41	0.596903381	0.597242849	0.613771986	0.616186083	0.61537179	0.607895218
43	0.581348019	0.595738472	0.59953611	0.604033462	0.624040095	0.600939232
45	0.58863518	0.597345042	0.601550547	0.612783771	0.622918867	0.604646681
47	0.585884069	0.587070831	0.587398384	0.601826001	0.619609974	0.596357852
49	0.589005108	0.588931988	0.593390743	0.603284343	0.616097709	0.598141978
51	0.579265517	0.5829962	0.583577224	0.592725988	0.595815501	0.586876086
53	0.592278442	0.5926817	0.603405578	0.60625988	0.61679315	0.60228375
55	0.596245225	0.595317325	0.597899678	0.606467503	0.616321917	0.60245033

Tabelle 3.7: Güte des Panoramabildes für die Testsequenz “Versuch 1 klein“ abhängig von den Parametern *log.rsz* und *log.cthresh*

Parameter log.rsz und log.NPTS

Die Ergebnisse der Untersuchung, wie sich die Güte des Panoramabildes durch die Variation der Parameter *log.rsz* und *log.NPTS* auswirkt, zeigt die Tabelle 3.8. Die höchste Güte pro Template-Seitenlänge (farblich markierte Zellen) für eine kleinformatige Sequenz liefert der *log.NPTS*-Parameterwert 6. Weiterhin ist ersichtlich, dass die Güte mit steigender Seitenlänge des Templates abnimmt (siehe Abbildung 3.9), sodass Templates mit der Seitenlänge von 19 bis 23 Pixel qualitative Panoramabilder liefern.

Bei großformatiger Testsequenz *Versuch 1 klein* gab es wiederum numerische Ausreißer

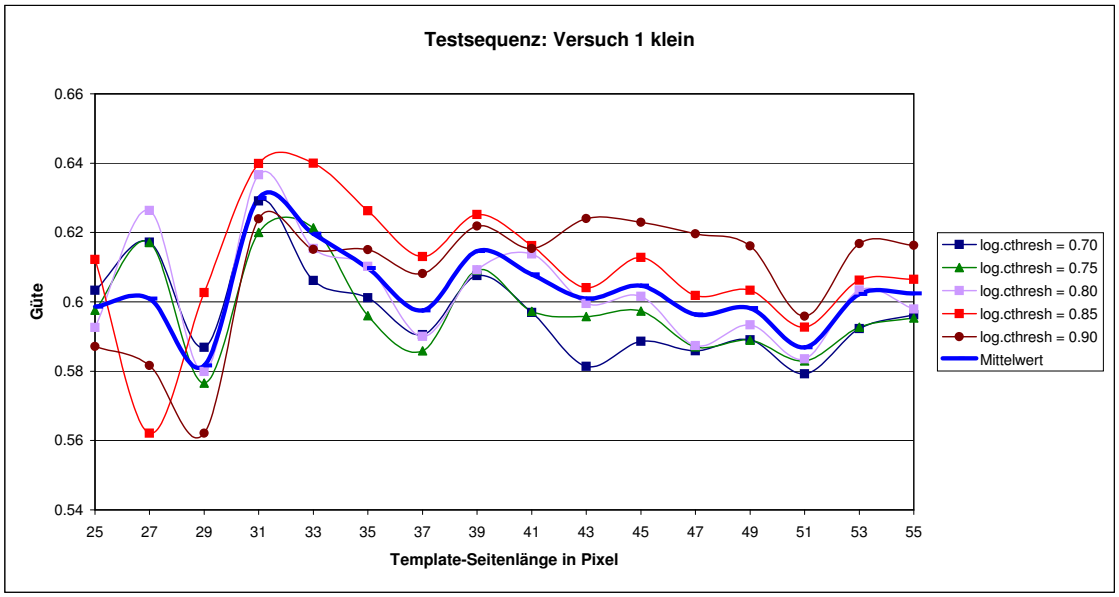


Abbildung 3.8: Güte des Panoramabildes für die Testsequenz "Versuch 1 klein" abhängig von den Parametern *log.rsz* und *log.cthresh*

bei den *log.NPTS*-Parameterwerten 6, 7 und 8, die in der Tabelle 3.9 farblich markiert und bei der Auswertung nicht berücksichtigt werden. Die Abbildung 3.10 illustriert die Güte der Panoramabilder für die Variation der zwei Parameter. So gibt es ein qualitativ hochwertiges Panoramabild bei einem Template mit der Seitenlänge von 31 Pixel. Die Anzahl der Spitzenwerte (Zellenmarkierungen) über alle Template-Seitenlängen verteilt, deuten auf bestmögliche Panoramabilder bei einem *log.NPTS*-Parameterwerten von 8 hin.

Anhand der Ergebnisse lässt sich der *log.NPTS*-Parameterwert bei kleinformatigen Bildsequenzen auf 6 eingrenzen, wogegen bei großformatigen Bildsequenzen ein Wert von 8 zu nehmen ist.

log.rsz	log.NPTS = 6	log.NPTS = 7	log.NPTS = 8	log.NPTS = 9	log.NPTS = 10	Mittelwert
15	0.771531794	0.7608315	0.743591685	0.777367164	0.732466997	0.757157828
17	0.763633331	0.749139655	0.741431553	0.764196063	0.751060311	0.753892183
19	0.769858239	0.74683513	0.747059979	0.767024514	0.751612009	0.756477974
21	0.76321624	0.738378431	0.755303457	0.759193312	0.74144106	0.7515065
23	0.761185322	0.741267127	0.754015172	0.750437121	0.75175532	0.751732012
25	0.753582882	0.740381806	0.76287423	0.755850469	0.751645038	0.752866885
27	0.753770353	0.746871123	0.753969018	0.748082291	0.751960999	0.750930757
29	0.761041655	0.748611099	0.760147544	0.754640553	0.755450045	0.755978179
31	0.756080589	0.75115947	0.74831652	0.74844507	0.751671938	0.751134717
33	0.732887123	0.742719374	0.757756764	0.747423607	0.758630125	0.747883399
35	0.742474221	0.746889772	0.753634355	0.738161794	0.751190249	0.746470078
37	0.738571465	0.748116196	0.752485682	0.744677137	0.752159605	0.747202017
39	0.734792977	0.740002382	0.741660935	0.742812376	0.750460276	0.741945789
41	0.747064887	0.740637988	0.750228897	0.752236147	0.758304616	0.749694507
43	0.739063958	0.748423031	0.751573936	0.738679341	0.749424349	0.745432923
45	0.749820552	0.736724427	0.744986257	0.741101814	0.745667262	0.743660063
47	0.747027801	0.751416259	0.747360631	0.74242882	0.746021079	0.746850918

Tabelle 3.8: Güte des Panoramabildes für die Testsequenz “Storz_parietal2“ abhängig von den Parametern *log.rsz* und *log.NPTS*

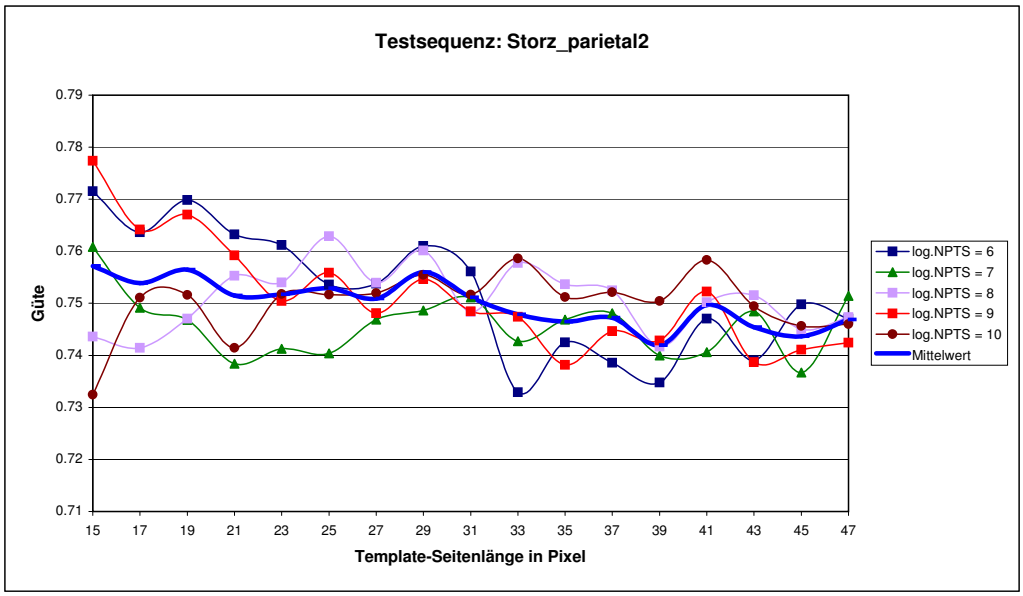


Abbildung 3.9: Güte des Panoramabildes für die Testsequenz “Storz_parietal2“ abhängig von den Parametern *log.rsz* und *log.NPTS*

log.rsz	log.NPTS = 6	log.NPTS = 7	log.NPTS = 8	log.NPTS = 9	log.NPTS = 10	Mittelwert
25	0.618385961	0.620423178	0.612178417	0.606495135	0.639164897	0.619329517
27	0.520954422	0.627319571	0.562080024	0.630733195	0.627923156	0.593802074
29	0.608403643	0.614443986	0.602681	0.628110878	0.613978434	0.613523588
31	0.641535226	0.6193867	0.639906687	0.630367386	0.60683088	0.627605376
33	0.572692591	0.60565605	0.639943945	0.594709047	0.605134206	0.603627168
35	0.638784075	0.574521028	0.626286515	0.596293242	0.613535149	0.609884002
37	0.631432272	0.6216441	0.61307358	0.612555592	0.603484261	0.616437961
39	0.592505838	0.60946716	0.625193897	0.605964386	0.602789254	0.607184107
41	0.637700346	0.611163308	0.616186083	0.574509505	0.587053314	0.605322511
43	0.608463839	0.615232821	0.604033462	0.5993961	0.603019977	0.60602924
45	0.615153142	0.59777284	0.612783771	0.604940347	0.610849174	0.608299855
47	0.627056136	0.590250701	0.601826001	0.596771621	0.615693887	0.606319669
49	0.624124167	0.598510282	0.603284343	0.605270274	0.608011287	0.60784007
51	0.613786615	0.595142671	0.592725988	0.606155669	0.600501677	0.601662524
53	0.610153383	0.593321957	0.60625988	0.593933699	0.60703092	0.602139968
55	0.61625717	0.605643643	0.606467503	0.592536291	0.612279855	0.606636893

Tabelle 3.9: Güte des Panoramabildes für die Testsequenz “Versuch 1 klein“ abhängig von den Parametern *log.rsz* und *log.NPTS*

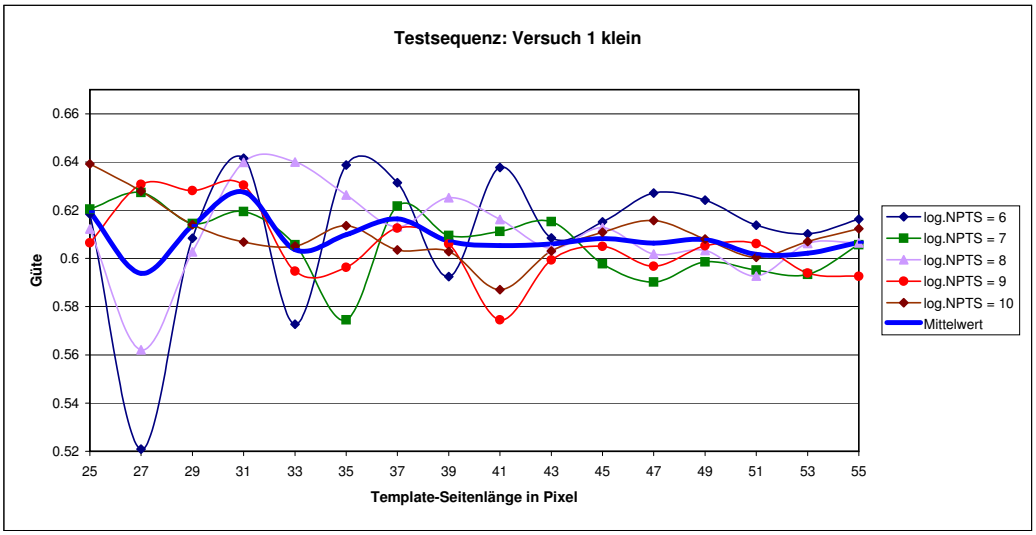


Abbildung 3.10: Güte des Panoramabildes für die Testsequenz “Versuch 1 klein“ abhängig von den Parametern *log.rsz* und *log.NPTS*

Kapitel 4

CUDA in Java

Angesichts des Problems, dass CUDA auf der Programmiersprache C, das ImageJ-Plugin hingegen auf Java aufsetzt, müssen beide Programmierwelten gekoppelt werden, um CUDA in Java zu benutzen.

Im Folgenden werden Konstrukte vorgestellt, die diese Sprachen verbinden können.

4.1 JNI

Durch JNI (Java Native Interface) lassen sich viele plattformspezifische Eigenschaften des Betriebssystems in Java nutzen. Zwar bewegt man sich mit Aufruf plattformspezifischer Funktionen vom plattformunabhängigen Ansatz von Java weg, ohne JNI würde man jedoch benötigte C-Bibliotheken zeitintensiv nachprogrammieren müssen.

JNI ermöglicht einem Programmierer in Java Routinen von Bibliotheken zu nutzen, die in C oder C++ geschriebenen sind. Des weiteren lässt sich aber auch die Java Virtual Maschine aus C nutzen. Mit dem Aufruf spezieller JNI Methoden ist ein Zugriff auf Java-Klassen deren Variablen, Methoden, und Objekten garantiert.

Mit Hilfe von JNI lassen sich somit die Programmiersprachen C bzw. C++ und Java koppeln.

Entwicklungsprozess

Der Entwicklungsprozess mit JNI, illustriert in Abbildung 4.1, lässt sich in folgende sechs Schritte gliedern[1]:

1. Schreiben der Java-Klasse

[1] Vgl. [SM02]

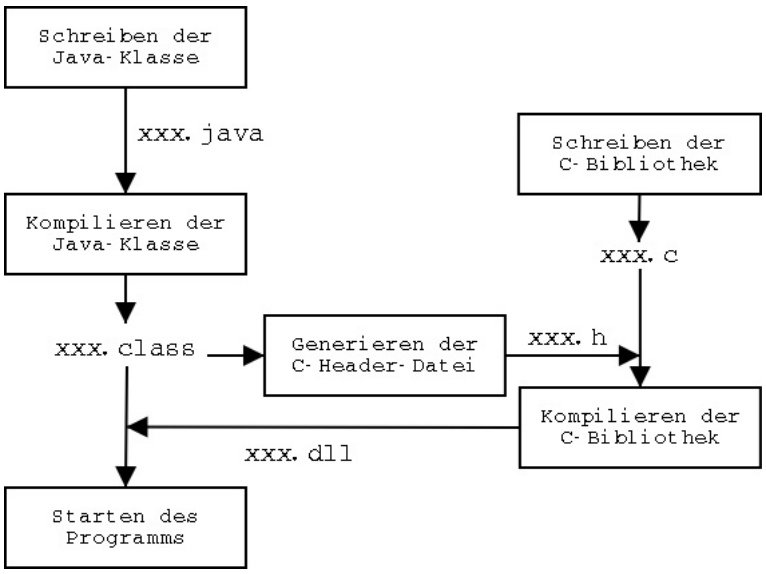


Abbildung 4.1: Typischer Entwicklungsprozess mit JNI

2. Kompilieren der Java-Klasse
3. Erzeugen einer C-Header-Datei aus der Java-Klasse (javah -jni Klassenname)
4. Schreiben der nativen C-Bibliothek mit entsprechenden C-Methoden
5. Kompilieren der C-Bibliothek
6. Starten der Java Klassen und Nutzung der nativen Methoden

Angemerkt sei, dass beim Schritt 5 eine dynamische Bibliothek für eine bestimmte Plattform und ein bestimmtes Betriebssystem erstellt wird. Unter Windows ist es z.B. eine DLL (Dynamic Link Library) unter Linux/Solaris eine Shared Library (*.so) usw.

Beispielprogramm

Das Vorgehen soll nun kurz an einem Beispiel erläutert werden. Listing 4.1 zeigt eine Java-Klasse *HelloWorld*. Die Anweisung in Zeile 3 *System.loadLybrary* bindet die Bibliothek *sayHelloWorld* zur Laufzeit ein, wobei hier nur der Name der Bibliothek ohne Endung angegeben wird. Der tatsächliche plattformspezifische Name wird beim Ladevorgang gesucht. Unter Windows wird beispielsweise nach *sayHelloWorld.dll* im gleichen

Verzeichnis wie die Java-Klasse und im PATH-Verzeichnis geschaut. Die Methode *print* wird mit Hilfe des Schlüsselwortes *native* als native Java Methode deklariert (Zeile 5). Aufruf der nativen Methode erfolgt in der Zeile 8.

Listing 4.1: Beispiel Java-Klasse mit Einbindung nativer Methode *print*

```
1  class HelloWorld {
2        static {
3            System.loadLibrary("sayHelloWorld");
4        }
5        private native void print();
6
7        public static void main(String[] args) {
8            new HelloWorld().print();
9        }
10 }
```

Listing 4.2: Native Methode *print* mit JNI in C

```
1  #include <jni.h>
2  #include <stdio.h>
3  #include <HelloWorld.h>
4  class HelloWorld {
5        JNIEXPORT void JNICALL Java_HelloWorld_print(JNIEnv *, jobject){
6            printf('Hello World from C!');
7        }
8  }
```

Durch eine Kompilierung der Java-Klasse und anschließenden Ausführung des Programms *javah* aus dem Java Development Kit (*javah -jni HelloWorld*), erhält man die Headerdatei *HelloWorld.h*. Sie deklariert die in C zu schreibende Methode: *JNIEXPORT void JNICALL Java_HelloWorld_print*.

Der C-Code in Listing 4.2 zeigt die Implementierung der nativen Methode *print*. In Zeilen 1 bis 3 werden nötige Headerdateien eingebunden. Zeile 5 beinhalten den Funktionsnamen, bestehend aus Java_Package_Klassenname_Funktionsname, gefolgt von einem Zeiger JNIEnr*, der auf das JNI-Environment verweist, und einem Pointer jobject, der eine Art This-Pointer auf das HelloWorld-Objekt selbst darstellt. In der nativen Methode erfolgt eine Konsolenausgabe (Zeile6).

Kompiliert man den C-Code und kopiert die daraus entstandene dll-Datei zu dem Java Programm *HelloWorld*, so lässt sich dieses starten und die native Methode nutzen.

Typ-Mapping

Das entscheidende Problem bei der Kommunikation zwischen C und Java stellen die unterschiedlichen primitiven Datentypen in beiden Sprachen dar. Diese werden wie in der Tabelle 4.1 dargestellt untereinander gemappt.

Java Typ	Nativer Typ	Größe in Bit
boolean	jboolean	8, unsigned
byte	jbyte	8
char	jchar	16, unsigned
short	jshort	16
int	jint	32
long	jlong	64
float	jfloat	32
double	jdouble	64
void	void	n/a

Tabelle 4.1: Primitive Java-Datentypen mit den korrespondierenden C-Datentypen in JNI

Für den Austausch von primitiven Variablen und Arrays zwischen C und Java stellt JNI Funktionen zur Verfügung. Beispielsweise lässt sich ein Zeiger auf ein Float-Array aus Java in C mit Hilfe der Methode *GetFloatArrayElements* setzen und mit *ReleaseFloatArrayElements* wieder lösen. Neben den Methoden für primitiven Datentypen bieten JNI auch Funktionen auf Objekte in einem Array. Allerdings werden dabei keine Zeiger verwendet, sondern Objektkopien. Dadurch wird aber in erster Linie die Performance der Anwendung beeinträchtigt.

4.2 Alternativen

Alternative Bindeglieder zwischen Java und CUDA sind JCublas und JaCuda. Beide Projekte sind OpenSource und sollen im folgenden kurz beschrieben werden.

JCublas

JCublas stellt Schnittstellen in Java für die BLAS[2]-Bibliothek von NVIDIA, auch als CUBLAS bezeichnet, zur Verfügung[3]. Die Bibliothek erlaubt einem Benutzer Vektor- und Matrizen-Multiplikationen auf CUDA kompatiblen Grafikkarten auszuführen. Aufgrund der beschränkten Nutzung von CUDA fällt diese Alternative zu JNI aus.

JaCuda

JaCuda beinhaltet zwei mathematische Funktionen, die auf CUDA kompatiblen Grafikkarten lauffähig sind[4]. Dazu gehören eine Matrizen-Multiplikation und eine Ähnlichkeitsberechnung von Massepunkten. Das Projekt befindet sich zur Zeit in Aufbau und ist noch im Alpha-Stadium. Diese JNI-Alternative fällt ebenfalls aufgrund einer Alpha-Version und der beschränkten Nutzung von CUDA aus.

[2] Basic Linear Algebra Subprograms
[3] Weitere Informationen unter [JCBL08]
[4] Siehe [JCD08] für nähere Informationen

Kapitel 5

Konzept zur Portierung rechenintensiver Java-Code-Teile in CUDA

Im ersten Abschnitt des Kapitels wird der rechenintensivste Teil des ImageJ-Plugins herausgearbeitet. Daraufhin wird dieser auf die Parallelisierbarkeit untersucht. Anschließend erfolgt die Erarbeitung eines Konzeptes zur Portierung der Java-Code-Teile in CUDA.

5.1 Identifizierung kritischer Code-Teile

Das ImageJ-Plugin zum Image Mosaicing wurde im Abschnitt 3.2 auf die Performance analysiert. Dabei stellte sich heraus, dass die Methode *crossCorrelation*, in der ein Kreuzkorrelationskoeffizient berechnet wird, am rechenintensivsten ist. Die Rechenintensität ist von der Seitenlänge eines Templates abhängig und steigt quadratisch mit der Template-Größe an.

Im Abschnitt 3.2.2 wurde die Methode *crossCorrelation* (Listing 3.2) optimiert. Auf der Basis soll nun die Portierung dieser in CUDA erfolgen.

5.2 Analyse der Code-Teile auf Parallelisierung

Der in Listing 3.2 optimierter Code dient als Ausgangspunkt für die Analyse. Parallelisierbar wären hierbei einzig und alleine die beiden for-Schleifen, in denen Zwischensummen für einen Kreuzkorrelationskoeffizient berechnet werden. (Zeile 8 bis 22). Der zweite Schritt bei der Berechnung des Kreuzkorrelationskoeffizienten in Zeile 24 und 25 erweist sich als schlecht parallelisierbar, da nur wenige Operationen parallel gemacht werden können und ein zu großer Overhead im Vergleich zu Operationen entstehen würde.

5.2.1 Dateneinsammlung

Zu den betreffenden Daten für eine Kreuzkorrelationskoeffizient-Berechnung gehören:

- **Referenzbild** - Aus diesem werden Pixelwerte für ein Template gelesen.
- **Aktuelles Bild** - Aus dem Bild werden Pixelwerte in einem bestimmten Bildausschnitt, der genauso groß ist wie ein Template, gelesen.
- **xy-Koordinate des Templates** - Die Koordinate stellt den Mittelpunkt des Templates im Referenzbild dar.
- **xy-Koordinate des aktuellen Bildausschnitts** - Die Koordinate bestimmt den Mittelpunkt eines Bildausschnitts im aktuellen Bild. Der Ausschnitt wird mit dem Template verglichen.
- **Breite und Höhe eines Templates** - Die Angabe ist in Pixel.
- **Offset** - Ein Offset beträgt der Hälfte einer Seitenlänge des Templates. Er wird für die Konstruktion eines Templates benötigt.
- **Breite des Referenzbildes** - Sie wird verwendet, um sich Zeilenweise über die Pixel im Referenzbild, sowie im aktuellen Bild zu bewegen.

Alle Daten außer dem Bildpaar liegen als primitive Datentypen vor, die problemlos in C über JNI zu transportieren wären. Die Bilddaten sind jedoch in einem Objekt vom Typ *ImageProzessor* untergebracht, und müssen vor dem Transport noch aufbereitet werden. Da die Grafikkarten mit Gleitkommazahlen einfacher Genauigkeit arbeiten, sollten die Pixelwerte der Bilder als eindimensionale Float-Arrays aus den Objekten extrahiert werden. Die passende Methode hierzu stellt *ImageProzessor* schon bereit.

5.2.2 Datenabhängigkeit

Die Abarbeitung der parallelen Programmteile auf den Grafikkarten erfolgt nach dem SIMD-Prinzip. Die Parallelität auf der Instruktionsebene kann aber nur erreicht werden, falls zwischen Instruktionen keine Datenabhängigkeit besteht.

In Listing 5.1 sind zwei for-Schleifen aus der optimierten Methode *crossCorrelation* zu sehen. Operationen in der äußeren for-Schleife (Zeile 3 und 4) weisen keine Abhängigkeiten auf. Das heißt, dass die Ergebnisse der Berechnungen in einer Schleifeniteration

unabhängig von Ergebnissen jeder anderer Schleifeniteration sind. Zwei Durchläufe der äußeren Schleife können also problemlos von zwei unterschiedlichen Threads parallel abgearbeitet werden. Operationen in der inneren for-Schleife weisen teilweise Abhängigkeiten auf. So können Pixel-Grauwerte aus den Pixel-Arrays problemlos unabhängig in unterschiedlichen Threads geholt und zwischengespeichert werden (Zeile 6 und 7). Die Produktberechnung in Zeilen 9, 11 und 12 kann ebenfalls in unterschiedlichen Threads erfolgen. Bei den fünf Summenberechnungen sieht das allerdings anders aus (Zeile 8 bis 12). Hier ist jeder Summenwert von den Iterationen davor abhängig.

In diesem kritischen Abschnitt muss in jeder Iteration erfolgender Schreibzugriff auf die Summen-Variablen synchronisiert werden, um eine Wettlaufsituation zwischen Threads, die darauf zugreifen, zu vermeiden.

Die Synchronisation der Addition ist zwar aus Korrektheitsgründen ein notwendiges Vorgehen, beeinflusst aber die Laufzeit des parallelen Programms negativ. Da das Endergebnis nicht von der Ausführungsreihenfolge der Einzeloperationen bei der Summenbildung abhängt, kann die Berechnung dennoch asynchron im so genannten Reduktionsverfahren[1] erfolgen.

Listing 5.1: Code-Ausschnitt aus der optimierten Methode *crossCorrelation*

```
1  ...
2   for (int v=−offs; v<=offs; v++) {
3           tRow = (v+ty)∗sizeXim + tx − offs;
4           iRow = (v+iy)∗sizeXim + ix − offs;
5           for (int u=−offs; u<=offs; u++) {
6                   valT = (0xFF & tpixels[tRow++]);
7                   valC = (0xFF & ipixels[iRow++]);
8                   sumT += valT;
9                   sumT2 += valT∗valT;
10                  sumC += valC;
11                  sumC2 += valC∗valC;
12                  prodCT += valT∗valC;
13          }
14  }
15 ...
```

[1] Siehe Abschnitt 5.3.4

5.3 Konzeption der CUDA-Methoden

5.3.1 Datentransport

Der Transport von Daten aus Java über JNI in C und anschließend in die Grafikkarte erfolgt auf folgende Weise:

Schritt 1

Jedes Bild eines Bildpaares wird in Java in ein eindimensionales Pixel-Array konvertiert, über JNI in C übertragen und anschließend in den Speicher der Grafikkkarte geladen.

Schritt 2

Alle übrigen Daten zur Berechnung eines Kreuzkorrelationskoeffizienten fließen aus Java über JNI in C als native Datentypen. Dort werden sie an den Kernel als Aufrufparameter übergeben, da dabei nicht die maximale Parametergröße von 256 Byte pro Kernel überstiegen wird.

Der zweite Schritt muss für jeden Kreuzkorrelationskoeffizient wiederholt werden, wobei die Anzahl der Datentransfers von dem Logarithmic-Search-Verfahren abhängt, der die Kreuzkorrelationskoeffizienten-Berechnung anstößt (Listing 3.4). In dem Code-Ausschnitt sind es bis zu neun Kreuzkorrelationskoeffizienten pro Suchmuster, die sequenziell abgearbeitet werden. Um hohe Performance auf dem Device zu erzielen, sollen die neun Kreuzkorrelationskoeffizienten parallel berechnet werden. Dazu müssen auf der Java-Seite die nötigen Daten gesammelt und an den Kernel weitergeleitet werden. Anschließend fließen die neun Ergebnisse ins Java zum Logarithmic-Search zurück.

Der erste und zweite Schritt sind jeweils pro Bildpaar zu wiederholen.

5.3.2 Datenspeicher

Die Wahl des Speichers für Daten im Device ist im wesentlichen entscheidend für die Dauer der Ausführung des Kernels, aufgrund von unterschiedlichen Latenzzeiten (siehe Tabelle 2.1).

Wie schon im Abschnitt 2.1.1 erläutert, besteht ein Device-Speicher aus drei Bereichen, die sich nicht alle für die aufkommenden Daten (Bildpaare) eignen. Die Größe der Float-Arrays eines Bildpaares hängt von der jeweiligen Auflösung ab. Bei Bildern

mit 360x288 Bildpunkten beträgt die Datengröße 0,395 MByte pro Bild und bei Bildern mit voller PAL-Auflösung (720x576) liegt sie bei 1,58 MByte pro Bild. Damit lässt sich Constant Memory mit seinem 64KByte kleinem Speicherbereich nicht nutzen. Texture Memory bzw. Global Memory bieten dagegen ausreichend Platz für mehrere Bilder. Die Latenzzeiten in den beiden Bereichen entsprechen 400 bis 600 Taktzyklen.

Während beim Global Memory die Latenzzeiten für Lese- und Schreibzugriffe durch Coalescing[2] bis zu zehnfach gemindert werden können, beträgt ein Zugriff beim Texture Memory im Falle eines Cache-Hits einem Takt. Somit fällt die Wahl für den Datenspeicher auf Texture Memory, der einen Cache aufweist und 2D Lokalitäten unterstützt.

5.3.3 Datenaufteilung

Eine effektive Nutzung aller zur Verfügung stehenden Multiprozessoren einer Grafikkarte kann nur durch eine optimale Aufteilung der Arbeit in Thread-Blöcke und der Verwendung von mehr Blöcken als MPs erreicht werden. Das heißt, dass im Falle einer Grafikkarte mit n Multiprozessoren die Rechenarbeit in mindestens n Blöcke zu je x Threads pro Block aufgeteilt werden muss. NVIDIA empfiehlt mindestens 64 Threads pro Block zu verwenden, besser jedoch 192 und mehr Threads pro Block, um die Latenzzeit bei Lese- und Schreibzugriffen auf den Grafikkartenspeicher zu verdecken[3]. Darüber hinaus sollte die Anzahl der Threads pro Block das Mehrfache von 32 sein, sodass die Threads in volle Warps gruppiert werden können, da ansonsten der letzte Warp jedes Blocks unnötigerweise mit wirkungslosen Threads gefühlt und abgearbeitet wird.

Die Datenmenge für einen Kreuzkorrelationskoeffizient hängt von der Seitenlänge eines Templates ab. Beträgt die Größe eines Templates beispielsweise 16x16 Pixel, so stellt sich die Datenmenge aus der Template-Größe und einem gleich großen Bildausschnitt aus dem aktuellen Bild zusammen und beträgt 256 Pixel pro Bild.

Für die Berechnung eines Kreuzkorrelationskoeffizienten müssen Grauwerte der Pixel aus den zwei Bildausschnitten paarweise geladen werden (siehe Listing 5.1 Zeile 6 und 7). In unserem Beispiel wären demnach 256 Threads nötig. Bei einer Blockgröße von 256 Threads pro Block (16x16) wäre nur ein Multiprozessor ausgelastet. Blockgrößen von 128 Threads pro Block (16x8) und 64 Threads pro Block (8x8) verteilen die Arbeit auf zwei beziehungsweise vier Multiprozessoren zur parallelen Ausführung. Nach [CUDAPG08]

[2] Coalescing bezeichnet einen optimierten Lese- oder Schreibzugriff im Global Memory. Dabei müssen die Daten in 64 Byte großen Regionen gelesen oder geschrieben werden.
[3] Siehe [CUDAPG08] S. 67

richtet sich die Wahl der Blockgröße je nach Problemstellung und muss nicht immer die höchste Anzahl an Threads pro Block betragen. Deshalb soll die optimale Blockgröße durch eine Messung der Rechenzeiten mit drei unterschiedlichen Blockgrößen ermittelt werden. Zum Einsatz kommen dabei Blöcke mit 64, 128 oder 256 Threads pro Block.

Angemerkt sei, dass bestimmte hardwaremäßige Grenzen bei Ressourcen einer Grafikkarte vorhanden sind. So darf die maximale Blockgröße von 512 Threads pro Block nicht überschritten werden. Des weiteren nutzt jeder Thread eine Registeranzahl für seine Berechnungen. Sie ist abhängig von dem Programm im Kernel. Die Threads teilen sich die maximale Anzahl an Registern (8192) eines Multiprozessors disjunkt. Das heißt, dass bei einer Blockgröße von 512 Threads pro Block und einer Nutzung von 17 Registern pro Thread 8704 Register zur Ausführung des Blocks auf einem Multiprozessor nötig wären. In dem Fall würde ein Kernel mit dieser Grid-Konfiguration erst gar nicht auf dem Device starten.

5.3.4 Kernel

Ein Kernel bekommt bei seiner Ausführung ein Grid zugeordnet, dessen Dimensionen erst zur Laufzeit festgelegt werden. Dadurch wird die Gesamtanzahl der Threads erst zur Ausführung des Kernels bestimmt. Die Konfiguration des Grids für den Kernel muss demnach variabel erfolgen. Zu verwenden sind jedoch zuvor ermittelte Blockgrößen von 64, 128 oder 256 Threads pro Block.

Während der Ausführung eines Kernels arbeitet jeder Thread jedes Blocks im Grid dasselbe Programm ab. Dabei besitzt jeder Thread eine Thread-ID und eine Block-ID, mit dessen Hilfe eine Iterationsvariable erzeugt werden kann, um die doppelte for-Schleife (Listing 5.1 Zeile 2 und 5) zu simulieren.

Im ersten Schritt des Kernels soll jeder Thread die Grauwerte eines Pixel-Paares aus den beiden Bildausschnitten holen (Zeile 6 und 7) und in Shared Memory ablegen. Da gleichzeitig neun Kreuzkorrelationskoeffizienten zu berechnen sind, werden demnach neun Pixel-Paare benötigt. Allerdings brauchen die Pixel aus dem Template nur einmal geladen werden, sodass pro Thread anstatt 18 nur 10 Ladevorgänge nötig sind.

Shared Memory soll als Zwischenspeicher für alle Berechnungen dienen, um die Latenzzeit bei Speicherzugriffen zu minimieren.

Im zweiten Schritt kann jeder Thread die Produktberechnungen (Zeile 9, 10 und 12) mit den zuvor in Shared Memory gespeicherten Grauwerten durchführen und das Ergeb-

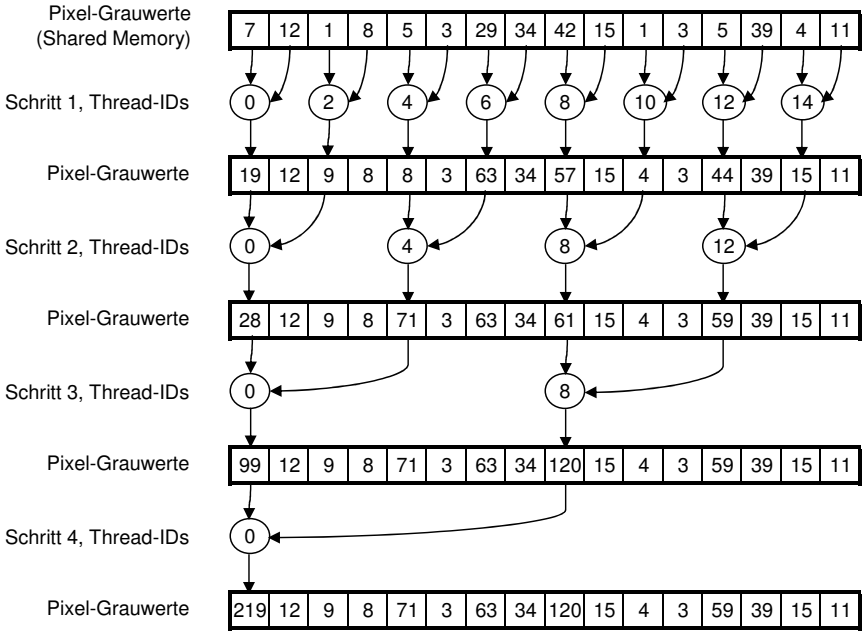


Abbildung 5.1: Reduktionsverfahren in CUDA am Beispiel von 16 Werten, erste Version

nis wiederum in Shared Memory zwischenspeichern. Anschließend muss an dieser Stelle eine Synchronisation der Threads in den Blöcken erfolgen, um vor dem nächsten Schritt (Summenbildung) sicher zu gehen, dass alle Pixel-Grauwerte geladen, deren Produkte gebildet und im Shared Memory abgelegt wurden.

Wie schon zuvor erwähnt, sollen die Summen (Zeile 8 bis 12) parallel anhand einer Reduktion im dritten Schritt des Kernels gebildet werden. Bei neun Kreuzkorrelationskoeffizienten muss die Aufsummierung der Werte aus dem Template (Zeile 8 und 9) nur einmal erfolgen. Somit sind insgesamt nur 29 Summen anstatt 45 zu bilden.

Ein Beispiel des Reduktionsverfahrens mit 16 Werten (halber Warp) ist in Abbildung 5.1 illustriert. Im ersten Schritt addiert jeder zweite Thread zwei benachbarte Werte und legt sein Ergebnis im Shared Memory ab. Diese Ergebnisse werden im nächsten Schritt aufsummiert und ebenfalls im Shared Memory abgelegt. Das Verfahren wird solange durchgeführt bis nur noch ein Ergebnis, das am Anfang eines Pixel-Grauwert-Arrays steht, übriggeblieben ist. Zwar gibt es bei der Reduktion keine Bank-Konflikte im Shared Memory, da jeder zweite Thread in einem halben Warp auf unterschiedliche Bänke

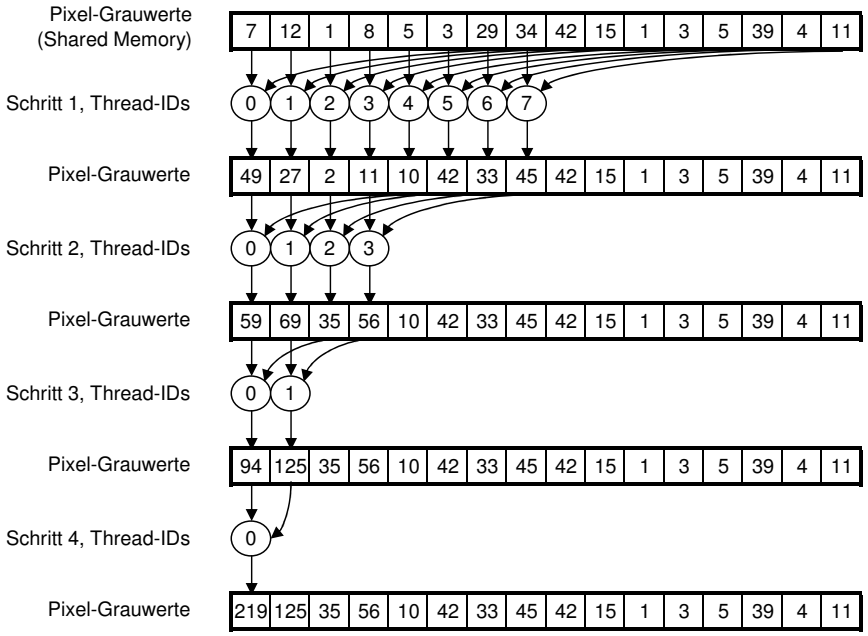


Abbildung 5.2: Reduktionsverfahren in CUDA am Beispiel von 16 Werten, zweite Version

zugreift, jedoch kommt es zu einer großen Divergenz in jedem Warp durch die Alternierung der Threads im halben Warp. Eine sequentielle Thread-Nutzung, wie im Beispiel in der Abbildung 5.2 zu sehen, vermindert die Divergenz in einem Warp. Alle arbeitenden Threads sind sequenziell angeordnet und addieren zwei Werte aus nicht benachbarten Bereichen. Dabei entstehen wiederum keine Bank-Konflikte im Shared Memory. Wie man dem Beispiel entnehmen kann, wird die Summe eines Arrays mit 16 Werten (2^4) parallel in 4 Schritten berechnet, im Gegensatz dazu wären bei einer sequenzieller Summenbildung 15 Schritte nötig. Daraus entgeht, dass beim Reduktionsverfahren auf einem Array mit 2^n Werten n Schritte bis zum Endergebnis notwendig sind.

Abschließend werden im vierten Schritt des Kernels die 29 Summenergebnisse aus dem Shared Memory in den Global Memory geschrieben.

Im Falle einer Arbeitsaufteilung auf mehrere Blöcke, handelt es sich bei den Summenergebnissen im Global Memory um Teilsummen, die noch vor einer Kreuzkorrelations-

koeffizient-Berechnung aufsummiert werden müssen. Die Summation der Teilsummen kann allerdings aufgrund der fehlenden Synchronisation auf der Blockebene nicht im gleichen Kernel erfolgen und sollte deshalb entweder im anderen Kernel oder auf der CPU durchgeführt werden. Je nach Blockgröße gibt es aber relativ wenige Teilsummen, deshalb werden die Teilsummen nach der Kernelausführung auf der CPU aufsummiert.

Kapitel 6

CUDA Implementierung

Im Zuge der Implementierung des konzipierten Kernels wurden drei unterschiedliche Blockgrößen getestet. Die schnellste Berechnung der neuen Kreuzkorrelationskoeffizienten war bei 64 Threads pro Block (8x8) zu verzeichnen. Deshalb beziehen sich alle folgenden Erläuterungen auf diese zweidimensionale Blockgröße.

Als Referenzgrafikkarte diente eine Geforce 8600 GT mit 256 MByte Arbeitsspeicher und 32 Prozessorkernen (4 Multiprozessoren) bei 1190 MHz.

Nachfolgende Beschreibung relevanter Änderungen im ImageJ-Plugin, sowie hiesiger Implementierung des Kernels wird bewusst ausführlich gehalten, um eine Einarbeitung für zukünftige Entwicklungen zu erleichtern.

6.1 Reengineering des ImageJ-Plugins

Das Plugin wurde um zwei native Methoden erweitert: *GPU_register_images* und *crossCorrelationGPUnineTemplate*. Sie befinden sich in einer DLL-Datei, die auch den Kernel für die Berechnung von neun Kreuzkorrelationskoeffizienten in sich trägt.

Die Steuerung der Methoden übernimmt die Java-Klasse des Plugins *Control_*. In dieser werden zunächst die Methoden-Köpfe deklariert (Listing 6.1 Zeile 2 bis 5) und die Bibliothek *LogSearchGPUnineCC* zur Laufzeit eingebunden (Zeile 7).

Der Transport von Bilddaten aus Java in C (Zeile 14) und anschließend in den Grafikkartenspeicher (Listing 6.3) erfolgt jedes mal nachdem ein konsekutives Bildpaar geladen und in eindimensionale Float-Arrays konvertiert wurde (Zeile 9 und 13). Die Methode *GPU_register_images* in Zeile 14 regelt den Transport von zwei Float-Arrays und zwei Bildauflösungsparameter über JNI.

Im weiteren Verlauf des ImageJ-Plugins beginnt das Logarithmic-Search-Verfahren in

Listing 6.1: Relevante Code-Erweiterungen in der Java-Klasse *Control_*

```
1  ...
2  public native static void GPU_register_images(float [] t_floatpixels, float [] c_floatpixels,
3                                                 int sizeXim, int sizeYim);
4  public native static double[] crossCorrelationGPUnineTemplate(int tx, int ty,
5                                                                 int[] vectors, int w, int h);
6
7  static {System.loadLibrary("LogSearchGPUnineCC");// Name der DLL-Datei (ohne .dll)}
8  ...
9  currfloat = (float[]) imp.getProcessor().convertToFloat().getPixels();
10 ...
11 for (int i =startfr+1;i<=endfr;i++){
12     ... //load next image
13     nextfloat = (float[]) colorNext.convertToFloat().getPixels();
14     GPU_register_images(currfloat, nextfloat, sizeX, sizeY);
15     ...
16     currfloat = nextfloat;
17 }
18 ...
```

Listing 6.2: Relevante Code-Erweiterungen in der Java-Klasse *LogSearch*

```
1  ...
2  int [] vectors = new int[9*2];
3  int k=0;
4  for (int i=0; i<9; i++) {
5          vectors[k++]= vec[i][0]-o; //cy
6          vectors[k++]= vec[i][1]-o; //cx
7  }
8
9  cvec = Control_.crossCorrelationGPUnineTemplate(t[1]-o,t[0]-o, vectors, 2*o+1, 2*o+1);
10 ...
```

den Methoden *search* und *ls_cross* der Java-Klasse *LogSearch* nach Bewegungsvektoren für bestimmte Templates aus dem Referenzbild im aktuellen Bild zu suchen. Bei der Suche wird ein Suchmuster mit neun Suchpunkten verwendet. Diese Punkte müssen an den Kernel weitergeleitet werden, um parallel neun Kreuzkorrelationskoeffizienten zu berechnen. Listing 6.2 zeigt die relevanten Änderungen in der Methode *ls_cross*. Dabei werden zuerst die xy-Koordinaten von den neun Punkten in einem eindimensionalen Integer-Array gesammelt (Zeile 4 bis 7). Anschließend werden die gesammelten Daten, sowie

die Template-Seitenlängen in Pixel der Methode *crossCorrelationGPUnineTemplate* als Parameter übergeben (Zeile 9). Nach der Berechnung der neuen Kreuzkorrelationskoeffizienten liefert die JNI-Methode ein eindimensionales Double-Array mit den Ergebnissen zurück. Die Werte werden zur weiteren Verarbeitung durch den Logarithmic-Search in der Variable *cvec* gespeichert.

6.2 JNI-Methoden

GPU_register_images

Listing 6.3 zeigt die Implementation der JNI-Methode *GPU_register_images* in C. In Zeile 8 und 9 werden Float-Zeiger auf eindimensionale Float-Arrays des Bildpaares über JNI-spezifische Methoden angelegt. Darauf folgend werden zwei Textur-Arrays auf dem Device über CUDA-spezifische Methoden alloziert (Zeile 17 und 18). Ihre Größe hängt von der Auflösung der Bilder im Bildpaar ab. Mit Hilfe der CUDA-Methode *cudaMemcpyToArray* wird der Inhalt der Float-Arrays in die Textur-Arrays kopiert (Zeile 19 und 20).

Letztendlich werden die Textur-Arrays an die zweidimensionalen Texturen, die global in Zeile 1 und 2 deklariert wurden, gebunden. Durch diesen Schritt liegen dann die Grauwerte des Bildpaares für den Kernel im Texture Memory zur Abholung bereit. Zum Schluss der Methode müssen noch die zuvor angelegten Float-Zeiger wieder gelöst werden.

crossCorrelationGPUnineTemplate

Die Implementation der JNI-Methode *crossCorrelationGPUnineTemplate* in C zeigt Listing 6.4. Die Methode steuert den Datenfluss zwischen Java und C, sowie Device und Host. Darüber hinaus führt sie den Kernel *textureCrossCorrelation9* zur Berechnung von neun Kreuzkorrelationskoeffizienten aus.

In Zeile 1 bis 3 werden Konfigurationsparameter des Kernels festgelegt. *dimBlock* stellt dabei die xy-Dimensionen eines Blocks dar und beträgt 64 Threads pro Block. Die Block-Anzahl in einem Grid hängt von den Seitenlängen des Templates ab und wird zur Laufzeit festgelegt (Zeile 2). Beträgt die Seitenlänge des Templates beispielsweise 16 Pixel, so gibt es pro Grid-Dimension zwei Blöcke, also vier Blöcke je 64 Threads.

In Zeile 3 wird Shared Memory pro Block festgelegt. Dabei berechnet sich die benötigte Anzahl an Bytes wie folgt. Ein Float besteht aus 4 Byte. Pro Thread werden 116 Byte

Listing 6.3: C-Code der JNI-Methode *GPU_register_images*

```
1  texture<float , 2> curImg;
2  texture<float , 2> refImg;
3
4  JNIEXPORT void JNICALL Java_Control_1_GPU_1register_1images(JNIEnv *env, jclass obj,
5          jfloatArray JNI_t_floatpixels, jfloatArray JNI_c_floatpixels, jint sizeX, jint sizeY){
6
7    jboolean isCopy = JNI_FALSE;
8    jfloat *tpixels = (jfloat*) env->GetPrimitiveArrayCritical(JNI_t_floatpixels, &isCopy);
9    jfloat *cpixels = (jfloat*) env->GetPrimitiveArrayCritical(JNI_c_floatpixels, &isCopy);
10
11   float *refArr = tpixels;
12   float *curArr = cpixels;
13
14   unsigned int bytes = sizeX * sizeY * sizeof(float);
15
16   cudaArray *cuRefArr, *cuCurArr;
17   cudaMallocArray(&cuRefArr , &refImg.channelDesc, sizeX, sizeY );
18   cudaMallocArray(&cuCurArr , &curImg.channelDesc, sizeX, sizeY );
19   cudaMemcpyToArray ( cuRefArr, 0, 0, refArr, bytes, cudaMemcpyHostToDevice );
20   cudaMemcpyToArray ( cuCurArr, 0, 0, curArr, bytes, cudaMemcpyHostToDevice );
21
22   cudaBindTextureToArray ( refImg , cuRefArr );
23   cudaBindTextureToArray ( curImg , cuCurArr );
24
25   env->ReleasePrimitiveArrayCritical(JNI_t_floatpixels, tpixels, JNI_ABORT);
26   env->ReleasePrimitiveArrayCritical(JNI_c_floatpixels, cpixels, JNI_ABORT);
27 }
```

benötigt. 8 Byte für den Pixel-Grauwert aus dem Template und seinen Wert zum Quadrat. Je 4 Byte für den jeweiligen Pixel-Grauwert aus den neun Bildausschnitten, sowie für dessen Quadratwert. Des weiteren wird für das Produkt eines Pixel-Grauwertes aus dem Template und dem Bildausschnitt je 4 Byte benötigt. Insgesamt belegt jeder Block 7424 Byte an Shared Memory in einem Multiprozessor.

In Zeile 7 wird auf dem Device in Global Memory Speicherplatz für die Teilergebnisse aus den Blöcken des Kernels reserviert. Nach einer Ausführung des Kernels mit zuvor festgelegter Konfiguration und den Übergabeparametern werden die Teilergebnisse des Kernels vom Device-Memory auf den Host-Memory kopiert (Zeile 13) und anschließend über alle Blöcke in der for-Schleife in Zeile 17 aufsummiert.

Als letztes werden die neun Kreuzkorrelationskoeffizienten mit den zuvor errechneten Summen nach der Gleichung 3.1 berechnet (Zeile 27) und die Resultate an Java zurückgegeben.

Listing 6.4: C-Code der JNI-Methode *crossCorrelationGPUnineTemplate*

```
1  dim3 dimBlock( 8, 8 );
2  dim3 dimGrid( templateWidth/dimBlock.x, templateHigh/dimBlock.y );
3  int smemSizeX = (2 + TemplateNumberX*3) * dimBlock.x * dimBlock.y * sizeof(float);
4
5  float *d_outdataX;
6  const int size_outdataX = (2+TemplateNumberX*3)* dimGrid.x * dimGrid.y * sizeof(float);
7  cudaMalloc( (void**)&d_outdataX, size_outdataX );
8
9  textureCrossCorrelation9<<<dimGrid, dimBlock, smemSizeX>>>
10  (d_outdataX, tx, ty, cxcy_vec[0], ..., cxcy_vec[17]);
11
12 float *h_outdataX = new float[size_outdataX];
13 cudaMemcpy( h_outdataX, d_outdataX, size_outdataX, cudaMemcpyDeviceToHost );
14
15 jdoubleArray result = env -> NewDoubleArray(TemplateNumberX);
16 jdouble* wert = (jdouble*) env->GetPrimitiveArrayCritical(result, JNI_FALSE);
17 for(int blockCounter=0; blockCounter<gridDimensionXY; blockCounter++){
18   for(m=0; m<TemplateNumberX; m++){
19         gpu_result_C[m] += h_outdataX[outdata_position++];
20         gpu_result_C2[m] += h_outdataX[outdata_position++];
21         gpu_result_TC[m] += h_outdataX[outdata_position++];
22   }
23   gpu_result_T += h_outdataX[outdata_position++];
24   gpu_result_T2 += h_outdataX[outdata_position++];
25 }
26 for(m=0; m<TemplateNumberX; m++){
27         wert[m]=(gpu_result_TC[m] - gpu_result_C[m] * gpu_result_T/pixelnumber)
28         /sqrt( (gpu_result_C2[m] - gpu_result_C[m]*gpu_result_C[m]/pixelnumber)
29         *(gpu_result_T2 - gpu_result_T*gpu_result_T/pixelnumber) );
30 }
31
32 return result;
33 }
```

6.3 Kernel

Der Kernel *textureCrossCorrelation9* soll wie im Konzept (Abschnitt 5.3.4) schrittweise erläutert werden.

Schritt 0: Variableninitialisierung

Wie schon zuvor erwähnt, werden mit Hilfe der Thread-IDs und der Block-IDs Iterationsvariablen simuliert und somit die doppelte for-Schleife (Listing 5.1, Zeile 2 und 5) realisiert.

Listing 6.5 zeigt die Variableninitialisierung im Kernel. In Zeilen 4 bis 11 werden die xy-Pixelkoordinaten für neun Bildausschnitte aus dem aktuellen Bild bestimmt. Analog dazu werden xy-Pixelkoordinaten aus dem Template des Referenzbildes in den Variablen *ref_x* und *ref_y* zwischengespeichert (Zeile 12 und 13). Darauf erfolgt die Bestimmung der Iterationsvariable für Shared Memory (Zeile 15), sowie die Reservierung des Shared Memory pro Block (Zeile 20).

Listing 6.5: Kernel-Code: Variableninitialisierung

```
1  #define TemplateNumberX 9
2  #define Blocksize 64
3  ...
4   int cur_x[TemplateNumberX];
5    cur_x[0] = cxcy_vec1 + threadIdx.x + blockIdx.x * blockDim.x;
6    ...
7    cur_x[8] = cxcy_vec17 + threadIdx.x + blockIdx.x * blockDim.x;
8   int cur_y[TemplateNumberX];
9    cur_y[0] = cxcy_vec0 + threadIdx.y + blockIdx.y * blockDim.y;
10   ...
11   cur_y[8] = cxcy_vec16 + threadIdx.y + blockIdx.y * blockDim.y;
12  int ref_x = tx + threadIdx.x + blockIdx.x * blockDim.x;
13  int ref_y = ty + threadIdx.y + blockIdx.y * blockDim.y;
14
15  int smem_i = threadIdx.y * blockDim.x + threadIdx.x;
16  int threadNumbers = blockDim.x * blockDim.y;
17  int offset = threadNumbers;
18  int blockCounter = gridDim.x * blockIdx.y + blockIdx.x;
19
20  extern __shared__ float sdata[];
21  float *sdataX;
22  sdataX = sdata;
```

Schritt 1 und 2: Pixel-Grauwerte

Das Laden der Pixel-Grauwerte aus zwei Texturen findet im Listing 6.6 in den Zeilen 2 und 9 statt. Sie werden in einer bestimmten Reihenfolge im Shared Memory gespeichert. Die Abbildung 6.1 veranschaulicht die Struktur eines Speicherbereiches im Shared Memory eines Blocks. Die Pixel-Grauwerte aus dem Template und deren Quadratwerte befinden sich am Ende des Shared Memory Speicherbereiches. Im vorderen Teil des Shared Memory alternieren jeweils drei Blöcke. Diese beinhalten die Pixel-Grauwerte aus dem Bildausschnitt, deren Quadratwerte, sowie das Produkt aus den Pixel-Grauwerten des aktuellen Bildes und des Templates.

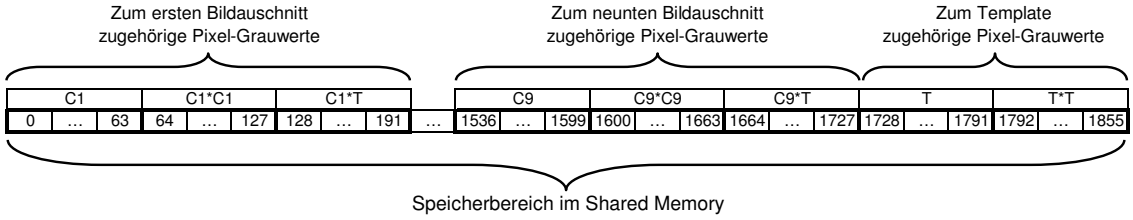


Abbildung 6.1: Pixel-Grauwert-Muster im Shared Memory eines Blocks

Listing 6.6: Kernel-Code: Laden von Pixel-Grauwerten

```
1  int positionPixelValueT = smem_i + offset * (3 * TemplateNumberX);
2  sdataX[positionPixelValueT] = tex2D(refImg, ref_x, ref_y); //PixelValueT
3  sdataX[positionPixelValueT + offset] =
4      sdataX[positionPixelValueT] * sdataX[positionPixelValueT]; //PixelValueT*PixelValueT
5
6  int position=0;
7  for(int m=0; m<TemplateNumberX; m++){
8      position = smem_i + offset*3*m;
9      sdataX[position] = tex2D(curImg, cur_x[m], cur_y[m]); //PixelValueC
10     sdataX[position + offset] = sdataX[position] * sdataX[position]; //PixelValueC*PixelValueC
11     sdataX[position + offset + offset] =
12             sdataX[position] * sdataX[positionPixelValueT]; //PixelValueC*PixelValueT
13 }
14 __syncthreads();
```

Schritt 3: Reduktionsverfahren

Die Aufsummierung der einzelnen Pixel-Grauwerte im Shared Memory erfolgt im Reduktionsverfahren, das im Listing 6.7 zu sehen ist. In der for-Schleife in Zeile 1 werden die Threads ermittelt, die zur späteren Bearbeitung benötigt werden. Dabei wird am Anfang die Anzahl der Threads auf die hälfte der Threads pro Block gesetzt, also auf 32. In jeder weiteren Iteration wird die Anzahl durch die schnelle Bit-Shift-Operation halbiert, solange die Anzahl größer null ist. Mit der if-Abfrage in Zeile 2 wird sichergestellt, dass nur zur Aufsummierung nötige Threads die Operationen im Schleifeninneren durchführen.

In Zeile 3 und 4 werden jeweils zwei Pixel-Grauwerte aus dem Template vom jeweiligen aktiven Thread zusammenaddiert. Analog dazu werden die Werte aus den neun Bildausschnitten des aktuellen Bildes aufsummiert.

Nach jeder Schleifeniteration müssen die Threads eines Blocks synchronisiert werden (Zeile 15).

Listing 6.7: Kernel-Code: Reduktion im Shared Memory

```
1  for(unsigned int s=threadNumbers/2; s>0; s>>=1) {
2    if (smem_i < s){
3      sdataX[positionPixelValueT] += sdataX[positionPixelValueT + s];
4      sdataX[positionPixelValueT + offset] += sdataX[positionPixelValueT + offset + s];
5      for(int m=0; m<TemplateNumberX; m++){
6              position = smem_i + offset*3*m;
7              //PixelValueC
8              sdataX[position] += sdataX[position + s];
9              //PixelValueCPow2
10             sdataX[position + offset] += sdataX[position + offset + s];
11             //PixelValueC*PixelValueT
12             sdataX[position + offset + offset] += sdataX[position + offset + offset + s];
13     }
14   }
15   __syncthreads();
16 }
```

Schritt 4: Sicherung der Ergebnisse

Listing 6.8 zeigt den Kernel-Code zur Abspeicherung der zuvor berechneten Summen in Global Memory.

In der for-Schleife in Zeile 2 bis 11 werden alle Summen der Pixel-Grauwerte, zugehörig

zu den neun Bildausschnitten, in Global Memory nacheinander abgelegt. Letztendlich kommen die Summen der Pixel-Grauwerte aus dem Template und deren Quadratwerte in den letzten Speicherbereich des Global Memory (Zeile 15 und 17).

Die im Global Memory abgespeicherten Ergebnisse werden nach der Ausführung des Kernels vom Host weiterverarbeitet.

Listing 6.8: Kernel-Code: Speichern der Ergebnisse im Global Memory

```
1  int outdata_position = 0;
2  for(int m=0; m<TemplateNumberX; m++){
3      outdata_position = TemplateNumberX*3*blockCounter + 3*m + blockCounter*2;
4      position = offset*3*m;
5      //Sum:PixelValueC
6      d_outdataX[outdata_position] = sdataX[position];
7      //Sum:PixelValueCPow2
8      d_outdataX[outdata_position + 1] = sdataX[position + offset];
9      //Sum:PixelValueC*PixelValueT
10     d_outdataX[outdata_position + 2] = sdataX[position + offset + offset];
11 }
12 positionPixelValueT = offset*(3*TemplateNumberX);
13 outdata_position += 3;
14 //Sum:PixelValueT
15 d_outdataX[outdata_position] = sdataX[positionPixelValueT];
16 //Sum:PixelValueT*PixelValueT
17 d_outdataX[outdata_position + 1] = sdataX[positionPixelValueT + offset];
```

6.4 Optimierung des Kernels auf Performance

Eine Optimierung der Performance eines Kernels um ca 10% liefert nach angaben von NVIDIA das sogenannte Entrollen. Damit ist die Auflösung von Schleifen mit fester Anzahl an Iterationen in CUDA gemeint. Eine Leistungssteigerung ist zu erwarten, da Adressarithmetik und Schleifenköpfe einen Overhead mit sich bringen.

In dem implementierten Kernel lassen sich Schleifen in Listing 6.6 Zeile 7, Listing 6.7 Zeile 1 und 5, sowie Listing 6.8 Zeile 2 entrollen, da zur Kompilierzeit die Iterationsanzahl jeder Schleife bekannt ist.

An dieser Stelle soll nur ein Beispiel gegeben werden, wie die zuvor genannten Schleifen im Kernel entrollt werden. Listing 6.9 zeigt die Entrollung der for-Schleife aus dem Listing 6.6 Zeile 7. Dabei wird der Schleifeninhalt für eine gegebene Anzahl an Iterationen

im Code wiederholt und die Iterationsvariable m (Zeile 1 und 8) fest definiert.

Eine weitere Optimierung der Performance bringt die Erhöhung der Operationen im Thread mit sich, da dadurch die Latenzzeiten für den Speicherzugriff minimiert werden. Die Erhöhung der Operationen pro Thread im implementierten Kernel konnte durch eine parallele Berechnung von bis zu 20 Kreuzkorrelationskoeffizienten erreicht werden. Die Grenze besteht auf Grund der maximalen Ausnutzung von Shared Memory Ressourcen eines Multiprozessors. So belegen 20 Kreuzkorrelationskoeffizienten 15872 Byte des 16 kByte großen Shared Memory.

Listing 6.9: Entrollen von Schleifen in CUDA

```
1  int m=0;
2  int position = smem_i + offset*3*m;
3  sdataX[position] = tex2D(curImg, cur_x[m], cur_y[m]); //PixelValueC
4  sdataX[position + offset] = sdataX[position] * sdataX[position]; //PixelValueCPow2
5  sdataX[position + offset + offset] =
6              sdataX[position] * sdataX[positionPixelValueT]; //PixelValueC*PixelValueT
7
8  ...
9
10 m=8;
11 position = smem_i + offset*3*m;
12 sdataX[position] = tex2D ( curImg, cur_x[m], cur_y[m]);
13 sdataX[position + offset]= sdataX[position]*sdataX[position];
14 sdataX[position + offset + offset] = sdataX[position]*sdataX[positionPixelValueT];
```

Kapitel 7

CUDA Ergebnis

Als Referenzgrafikkarten zur Messung der Ergebnisse dienten zwei Geforce Grafikkarten, deren Spezifikationen in der Tabelle 7.1 zusammengefasst sind.

Produkttyp:	**Geforce 8600 GT**	**Geforce 8800 GT**
Arbeitsspeicher :	256 MB DDR2	512 MB DDR3
Speichertakt:	750 MHz	900 MHz
Speicherschnittstelle:	128 Bit	256 Bit
Multiprozessoren:	4	14
MP Takt:	540 MHz	600 MHz
Prozessorkerne:	32	112
PK Takt:	1190 MHz	1500 MHz

Tabelle 7.1: Grafikkartenspezifikationen

Tabelle 7.2 stellt die Berechnungszeiten pro Kreuzkorrelationskoeffizient in CUDA gegenüber den C-Versionen[1] abhängig von der Seitenlänge eines Templates dar, wobei die Datentransportzeiten mit eingerechnet sind.

Betrachtet man die Geschwindigkeiten der CPUs, so spiegelt sich die ca. zweifache Verdoppelung der Rechenleistung in GHz in den Berechnungszeiten wieder. Etwas anders sieht es bei den Grafikkarten aus. Hier spielt nicht nur die Rechenleistung und die Anzahl der Multiprozessoren eine wichtige Rolle, sondern auch die Arbeitsaufteilung (Blöcke) pro Templategröße. Aber auch die Ressourcen eines Multiprozessors, die für den Scheduler mehr Freiheiten einräumen, sind nicht zu verachten. Die genannten Eigenschaften schlagen sich in den GPU-Rechenzeiten nieder, wie die Abbildung 7.1 veranschaulicht.

Wie schon zuvor erwähnt, verbraucht ein Kernel mit 20 Kreuzkorrelationskoeffizienten-Berechnungen 15872 Byte an Shared Memory pro Block (Spalten: *4MP_20T_pro_Kernel*,

[1] Siehe Abschnitt 3.2.3

	Rechenzeiten pro Kreuzkorrelationskoeffizient in Nanosekunden					
Template-Seitenlänge in Pixel	Pentium M 1,4 GHz	Athlon64 X2 2,5GHz	8600GT (4 Multiprozessoren)		8800GT (14 Multiprozessoren)	
log.rsz	**1,4GHz_JNI C**	**2,5GHz_JNI C**	**4MP_9T_pro_Kernel**	**4MP_20T_pro_Kernel**	**14MP_9T_pro_Kernel**	**14MP_20T_pro_Kernel**
15	4048	2019	9486	4647	6800	3487
23	6515	3280	12457	10488	6891	3576
31	10112	5127	12742	13431	7083	5652
39	14459	7439	18545	25164	7122	13506
47	19704	9978	41501	30588	26486	15649
63	31865	16153	51431	44111	28891	20078
95	73592	36400	81801	105298	36682	33100
127	128940	63721	123887	187715	46986	50583
159	200105	102383	176789	287928	59826	72704
191	287619	142093	251560	397501	75398	102831
255	528275	260292	421029	749132	115847	174029

Tabelle 7.2: Rechenzeiten pro Kreuzkorrelationskoeffizient in CUDA und in C im direkten Vergleich

14MP_20T_pro_Kernel). Dadurch kann nur ein Block pro Multiprozessor aktiv werden. Bei neun Kreuzkorrelationskoeffizienten-Berechnungen werden nur 7424 Byte an Shared Memory pro Block verwendet (Spalten: *4MP_9T_pro_Kernel*, *14MP_9T_pro_Kernel*). Das erlaubt dem Scheduler zwei Blöcke pro Multiprozessor parallel zu betreiben, sodass 128 Threads pro Multiprozessor aktiv sind. Dies wird bei beiden Grafikkarten ab einer Template-Seitenlänge von 39 Pixel deutlich. Die kleinere Anzahl an Kreuzkorrelationskoeffizienten-Berechnungen pro Kernel weisen gegenüber größeren Anzahlen schnellere Rechenzeiten auf. Bis zur Templategröße von 39x39 Pixel kann der Scheduler, aufgrund der geringen Blockanzahl, seine Arbeit nicht effizient verrichten. Beispielsweise wird die Rechenarbeit bei einem Template mit 16x16 Pixel auf vier Blöcke mit je 64 Threads aufgeteilt. Das beschäftigt gleichzeitig vier Multiprozessoren. Die Geforce 8600 GT ist dabei voll ausgelastet. Die Geforce 8800 GT hat allerdings noch 10 weitere Multiprozessoren unbeschäftigt. Trotzdem ist die unterforderte Grafikkarte etwas schneller bei beiden Konfigurationen des Kernels, weil die Taktung des Speichers, MPs und der Prozessorkerne geringfügig schneller ist.

Die deutlichen Sprünge in den Berechnungszeiten der beiden Grafikkarten im Template-Seitenlängenbereich von 31 bis 47 Pixel (siehe Abbildung 7.1) sind auf den Scheduler zurückzuführen. Beispielsweise kann der Scheduler bei der Geforce 8800 GT bis zu einer Template-Seitenlänge von 39 Pixel (25 Blöcke) die Rechenzeiten fast auf einem Niveau durch die gleichzeitige Handhabung der 25 Blöcke auf 14 Multiprozessoren halten (Spalte: *14MP_9T_pro_Kernel*). Ab einer Template-Seitenlänge von 47 Pixel gibt es mehr Blöcke als der Scheduler gleichzeitig auf den Multiprozessoren aktiv schalten kann, was

zum Anstieg der Rechenzeit führt.

Mit dem implementierten Kernel lassen sich die Berechnungszeiten einer 2,5 GHz CPU erst ab 95 Pixel pro Template-Seitenlänge mit einer Grafikkarte mit 14 Multiprozessoren übertreffen. So gibt es eine deutliche Beschleunigung bei den Kreuzkorrelationskoeffizienten-Berechnungen durch eine GPU gegenüber einer CPU bei 255 Pixel pro Seitenlänge eines Templates. Der Beschleunigungsfaktor liegt bei ca. 2,24 gegenüber einer 2,5 GHz CPU und bei ca. 4,56 gegenüber einer 1,4 GHz CPU. Mit Hilfe einer Geforce der neusten Generation mit 30 Multiprozessoren lassen sich die Beschleunigungsfaktoren theoretisch verdoppeln, da der Kernel skalierbar ist.

Die Beschleunigungsfaktoren von Kreuzkorrelationskoeffizienten-Berechnungen auf der GPU fallen im Vergleich zu Matrixberechnungen, die Ähnlichkeiten in der Problemstellung und deren Lösung aufweisen, relativ niedrig aus. Das liegt vor allem daran, dass bei den Kreuzkorrelationskoeffizienten-Berechnungen das Fünffache an Shared Memory verwendet wird und somit der Spielraum für mehr aktive Threads pro Multiprozessor eingeschränkt ist.

Im Rahmen des ImageJ-Plugins, das für Bildsequenzen mit einer halben und vollen PAL-Auflösung Templates mit einer Seitenlänge zwischen 15 und 45 Pixel für ein qualitatives Matching verwendet, konnten keine Beschleunigung bei den Kreuzkorrelationskoeffizienten-Berechnungen erreicht werden.

Bei der zukünftigen Verarbeitung von aktuellen Formaten, wie dem *Medical HD*[2], sind größere Templates als 45x45 Pixel notwendig, um qualitative Panoramabilder zu erzielen. Erst da könnte sich die Grafikkarte als Co-Prozessor im ImageJ-Plugin beweisen.

[2] Auflösung: 1280x1024 Pixel

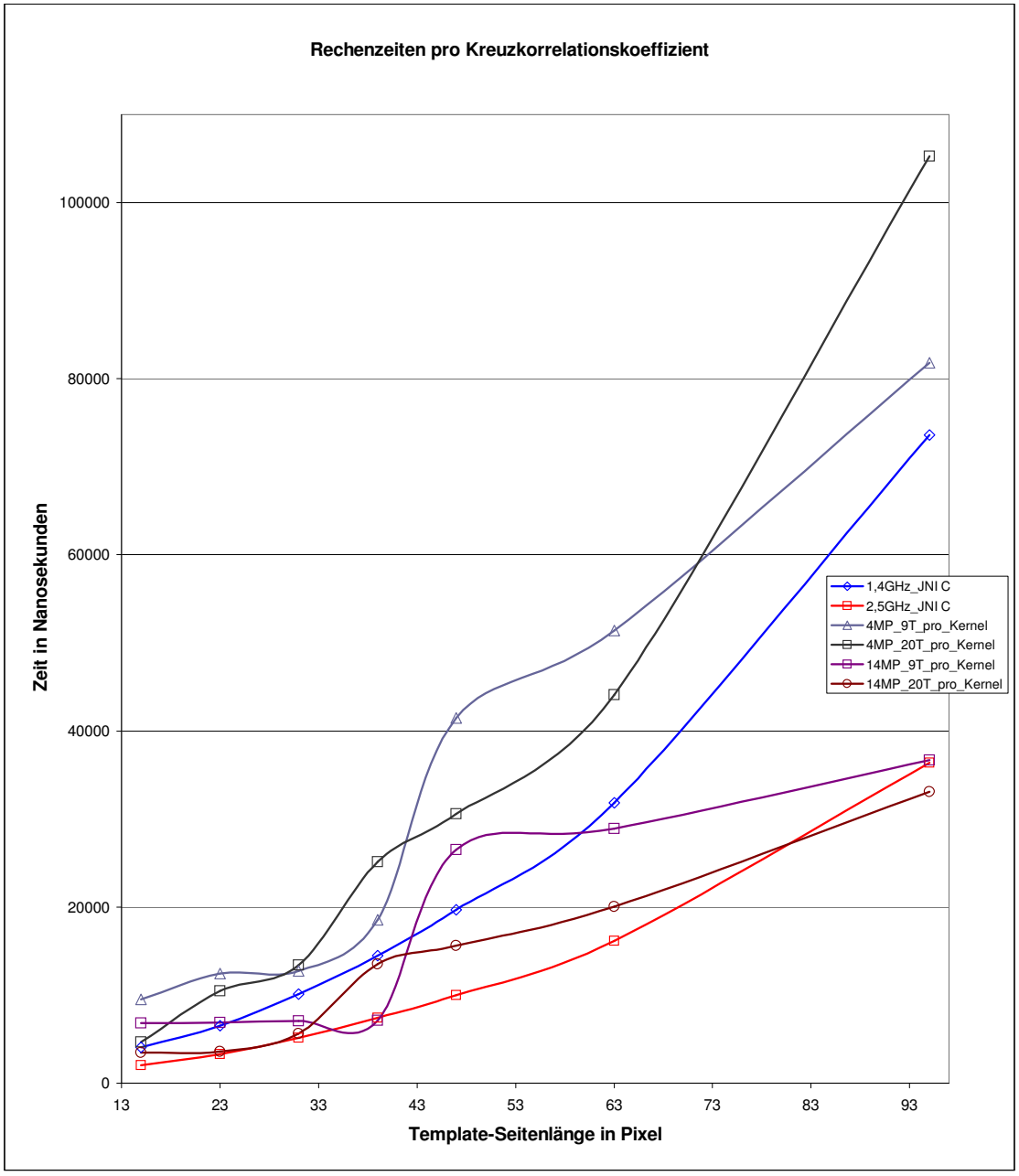


Abbildung 7.1: Gegenüberstellung der Rechenzeiten pro Kreuzkorrelationskoeffizient in CUDA und in C

Kapitel 8

Fazit und Ausblick

8.1 Fazit

Im Zuge dieser Arbeit wurde das bestehende Logarithmic-Search-Verfahren zum Image Mosaicing im ImageJ-Plugin auf Performance und Qualität hin optimiert. Im Vordergrund stand die Optimierung der Performance nicht nur auf einer CPU, sondern auch auf einer GPU mit Hilfe des CUDA-Frameworks von NVIDIA.

Im Vergleich zur ersten, unoptimierten CPU-Version, wurde eine Beschleunigung auf der CPU um den Faktor 2 erreicht. Bei zuhilfenahme einer GPU konnte auf Grund der hohen Latenzzeiten beim Speicherzugriff und der geringen Ressourcen in einem Multiprozessor keine Beschleunigung für die spezielle Problemstellung im ImageJ-Plugin erzielt werden.

Um die Einarbeitung für zukünftige Arbeiten zu erleichtern, sind die wichtigsten Änderungen in den Methoden des ImageJ-Plugins beschrieben und erläutert worden.

Die Frage „Wie gut gelingt es, typische Algorithmen der Bildverarbeitung auf ein paralleles Framework abzubilden?“ kann nicht eindeutig beantwortet werden. Einerseits wurde für die Lösung der speziellen Problemstellung im ImageJ-Plugin keine Beschleunigung erzielt. Andererseits konnte gezeigt werden, dass Beschleunigungen bei gleicher Problemstellung im größeren Umfang auf einem parallelen Framework durchaus zu erreichen sind.

8.2 Ausblick

Weitere Untersuchungen könnten sich mit der Parametrisierung des implementierten Vergleichsmaßes auseinander setzen. Dabei ist zu untersuchen, ob die nach [Bloem09]

empfohlene Konfiguration von 5x5 Templates die optimale Einstellung für kleinformatige, sowie großformatige Bildsequenzen ist. Des weiteren wäre mit dem Vergleichsmaß zu untersuchen, in wie weit sich unterschiedliche Kombinationen der zahlreichen Parameter des Logarithmic-Search-Verfahrens auf die Qualität des Panoramabildes auswirken. Dieses Thema konnte im Rahmen der Arbei nicht weitreichend ausgearbeitet werden.

Weitere Optimierungen in der Performance und der Qualität des Logarithmic-Search-Verfahrens könnten andere Suchmuster[1] oder Suchstrategien[2] liefern.

Durch kürzlich erschienenen OpenCL[3] Standard, der Berechnungen auch auf nicht CUDA-fähigen Grafikkartenarchitekturen erlaubt, könnte geprüft werden, ob andere Grafikkarten mit größeren Multiprozessorressourcen zur Beschleunigung des Logarithmic-Search-Verfahrens nutzbar wären.

Wie im Abschnitt 3.2.2 gezeigt, könnten weitere Code-Teile des ImageJ-Plugins in die Programmiersprache C überführt werden, um die Performance der Anwendung zu steigern. Darüber hinaus kann die Abarbeitung der Code-Teile in C mit Hilfe von OpenMP[4] auf mehrere CPUs verteilt werden. Hierbei wäre zu untersuchen, ab welcher Thread-Menge oder Arbeitsmenge eine Beschleunigung auf den Multi-Core-Prozessoren gegenüber der Single-Core-Prozessoren statt findet und ob sich der Einsatz von OpenMP in dem ImageJ-Plugin lohnt.

[1] z.B. [Lundmark01]
[2] z.B. [PNWC08], [JTG08]
[3] Open Computing Language, weitere Informationen unter: `http://www.khronos.org/opencl/`
[4] Für weitere Informationen: `http://openmp.org`

Kapitel 9

Literaturverzeichnis

[Adobe08] Adobe: *GPU Acceleration support in Acrobat and Adobe Reader (8.x)*, TechNote, 2008, http://www.adobe.com/cfusion/knowledgebase/index.cfm?id=333447, abgerufen am 01.12.2008

[AMD08] AMD: *ATI Stream Technology*, 2008, http://ati.amd.com/technology/streamcomputing/index.html, abgerufen am 01.02.2009

[BB05] Wilhelm Burger, Mark James Burge: *Digitale Bildverarbeitung - Eine Einführung mit Java und ImageJ*, Springer-Verlag Berlin Heidelberg, 2005

[Behrens08] Alexander Behrens: *An Image Mosaicing Algorithm for Bladder Fluorescence Endoscopy*, RWTH Aachen, 2008, http://www.lfb.rwth-aachen.de/files/publications/2008/BEH08a.pdf, abgerufen am 22.11.2008

[BFGS03] Jeff Bolz, Ian Farmer, Eitan Grinspun, Peter Schröder: *Sparse matrix solvers on the GPU: conjugate gradients and multigrid*, ACM SIGGRAPH, 2003, http://www.multires.caltech.edu/pubs/GPUSim.pdf, abgerufen am 01.12.2008

[Bloem09] Jannis Bloemendal: *Schnelle Verfahren zur Objektregistrierung in der Bildverarbeitung am Beispiel der Gesichtsstabilisierung*, Diplomarbeit, Februar 2009, http://www.gm.fh-koeln.de/~konen/Diplom+Projekte/PaperPDF/DA-Bloemendal09.pdf, abgerufen am 24.03.2009

[CBDR03] Philippe Colatoni, Nabil Boukala, Jérôme Da Rugna: *Fast and Accurate Color Image Processing Using 3D Graphics Cards*, 8th International Fall Workshop on Vision, Modeling and Visualization (VMV 2003), 2003, http://colantoni.nerim.net/articles/VMV2003.pdf, abgerufen am 01.12.2008

[CUDAPG08] NVIDIA: *CUDA 2.0 Programming Guide*, 2008. http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf, abgerufen am 30.11.2008

[FVLS04] B. Fischer, B. Vaessen, T. E. Lehmann, K. Spitzer: *Bildverbesserung endoskopischer Videosequenzen in Echtzeit*, Institut für medizinische Informatik, Universitätsklinikum RWTH Aachen, 2004, http://phobos.imib.rwth-aachen.de/lehmann/ps-pdf/BVM2004_528.pdf, abgerufen am 22.11.2008

[FWKL05] Zhe Fan, Xiaoming Wei, Arie Kaufman, Wei Li: *GPU-Based Flow Simulation with Complex Boundaries*, GPU Gems II: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley, 2005, http://www.cs.sunysb.edu/~vislab/projects/amorphous/WeiWeb/HardwareLBM/gpu_flow.pdf, abgerufen am 01.12.2008

[GGZ06] Alexander Greß and Gabriel Zachmann: *GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures*, Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS 06), Rhodes Island, Greece, 2006, http://cg.in.tu-clausthal.de/papers/gpu-abisort-ipdps-2006/gpu-abisort-ipdps-2006.pdf, abgerufen am 01.12.2008

[Golem08] Adobe: *Neue Photoshop-Versionen setzen auf GPU-Unterstützung* , Artikel 26.05.2008, 2008, http://www.golem.de/0805/59939.html, abgerufen am 01.12.2008

[GRHTM05] Naga K. Govindaraju, Nikunj Raghuvanshi, Michael Henson, David Tuft, Dinesh Manocha: *A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors*, Uni-

versity of North Carolina at Chapel Hill, 2005, http://gamma.cs.unc.edu/SORT/gpusort.pdf, abgerufen am 01.12.2008

[HT4U08] HT4U: *Intel Larrabee - Quick Preview*, Artikel 04.08.2008, http://ht4u.net/reviews/2008/larrabee_preview/, abgerufen am 01.02.2009

[JCBL08] Javagl: *JCublas* , http://javagl.de/jcuda/jcublas/JCublas.html, abgerufen am 02.02.2008

[JCD08] Gert Wohlgemuth: *JaCuda* , http://sourceforge.net/projects/jacuda, abgerufen am 02.02.2008

[JKW09] J. Schneider, M. Kraus, R. Westermann: *GPU-Based Real-Time Discrete Euclidean Distance Transforms With Precise Error Bounds*, VISAPP 2009, 2009, http://wwwcg.in.tum.de/Research/data/Publications/visapp09.pdf, abgerufen am 01.12.2008

[JTG08] C. Jeong, T. Ikenaga, S. Goto: *An Extended Small Diamond Search Algorithm For Fast Block Motion Estimation*, Waseda University, Japan, 2008 http://www.ieice.org/explorer/ITC-CSCC2008/pdf/p1037_P1-8.pdf, abgerufen am 31.03.2009

[KBS07] Wolfgang Konen, Beate Breiderhoff, Martin Scholz: *Real-Time Image Mosaic for Endoscopic Video Sequences*, Springer Verlag Heidelberg, Bildverarbeitung für die Medizin 2007, 2007 http://www.springerlink.com/content/q2556078gp3763j5/fulltext.pdf, abgerufen am 22.11.2008

[Konen06] Wolfgang Konen: *Optischer Fluss und Echtzeit-Videobearbeitung*, Technical Report, Institut für Informatik, FH Köln, 2006, http://www.gm.fh-koeln.de/~konen/WPF-BV/TR-OpticalFlow-ImaMos.pdf, abgerufen am 30.11.2008

[Kourogi99] M. Kourogi, T. Kurutut, J. Hoshinot, Y. Mumoka: *Real-time image mosaicing from a video sequence*, Procs ICIP99, vol. 4, 133-137, 1999, http://www.is.aist.go.jp/kurata/demo/distribution/1999/icip99.pdf, abgerufen am 22.11.2008

[KW03] Jens Krüger, Rüdiger Westermann: *Linear algebra operators for GPU implementation of numerical algorithms*, ACM Transactions on Graphics (TOG), ACM SIGGRAPH, 2003, http://wwwcg.in.tum.de/Research/data/Publications/sig03.pdf, abgerufen am 01.12.2008

[LOGKHLT05] David Luebke, John D. Owens, Naga Govindaraju, Jens Krüger, Mark Harris, Aaron E. Lefohn, Timothy J. Purcell: *A Survey of General-Purpose Computation on Graphics Hardware*,State of The Art Report, EUROGRAPHICS, 2005, http://www.idav.ucdavis.edu/func/return_pdf?pub_id=844, abgerufen am 24.11.2008

[Lundmark01] Astrid Lundmark: *Non-Redundant Search Patterns in Log-Search Motion Estimation*, Image Coding Group, Department of Electrical Engineering, Linköping University, SSAB Symposium on Image Analysis, 2001, http://www.icg.isy.liu.se/publications/en/SSAB2001Lundmark.pdf, abgerufen am 31.03.2009

[MDP06] Sebastien Mazare, Jean-Luc Dugelay, Renaud Pacalet: *USING GPU FOR FAST BLOCK-MATCHING*, EUSIPCO 2006, 14th European Signal Processing Conference, Florence, Italy, 2006, http://www.eurecom.fr/util/publidownload.en.htm?id=1972, abgerufen am 01.12.2008

[Naderi07] Martin Naderi: *Implementierung eines Echtzeitverfahrens zur Erstellung von Bildmosaiken aus endoskopischen Videosequenzen*, Bachelorarbeit, 2007, http://www.gm.fh-koeln.de/~konen/Diplom+Projekte/PaperPDF/BA_Naderi_03.04.07.pdf, abgerufen am 22.11.2008

[Netzwelt08] Netzwelt: *Video-Konverter mit Geforce-Turbo*, Artikel 14.08.2008, http://www.netzwelt.de/software/8644-badaboom.html, abgerufen am 28.11.2008

[OLGHKLP07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, Timothy J. Purcell: *A Survey of General-Purpose Computation on Graphics Hardwares*, Computer Gra-

phics Forum, Volume 26, 2007, http://stage.jordyvaneijk.nl/gpgpuSurvey2007.pdf, abgerufen am 01.12.2008

[PNWC08] Lai-Man Po, Ka-Ho Ng, Ka-Man Wong, Kwok-Wai Cheung: *Multi-Direction Search Algorithm For Block-based Motion Estimation*, City University of Hong Kong, China, 2008, http://www.ieice.org/explorer/ITC-CSCC2008/pdf/p1037_P1-8.pdf, abgerufen am 31.03.2009

[Schiwietz08] Thomas Schiwietz: *Acceleration of Medical Imaging Algorithms Using Programmable Graphics Hardware*, PhD Thesis, Technischen Universität München, 2008, http://mediatum2.ub.tum.de/download/653736/653736.pdf, abgerufen am 01.12.2008

[SPM02] Satya Prakash Mallick: *Feature Based Image Mosaicing*, Department of Electrical and Computer Enginering, University of California, San Diego, 2002. http://www-cse.ucsd.edu/classes/fa02/cse252c/smallick.pdf, abgerufen am 28.11.2008

[StemmerI07] Stemmer Imaging: *Bildverarbeitung auf der Grafikkarte: GPU schlägt CPU*, Newsarticle 09.10.2007, http://www.imaging.de/pages/news/item.php?view=319&item=1300, abgerufen am 28.11.2008

[SM02] Sun Microsystems: *Java Native Interface: Programmer's Guide and Specification*, http://java.sun.com/docs/books/jni/, abgerufen am 10.02.2008

[SzSh97] Richard Szeliski and Heung-Yeung Shum: *Creating Full View Panoramic Image Mosaics and Environment Maps*, Computer Graphics (SIGGRAPH'97 Proceedings), 1997, http://portal.acm.org/citation.cfm?id=258861&dl=GUIDE&coll=GUIDE, abgerufen am 28.11.2008

[VKNS03] Florian Vogt, Sophie Krüger, Heinrich Niemann, Christoph Schick: *A System for Real-Time Endoscopic Image Enhancement*, Springer Verlag Heidelberg, Volume 2879/2003, 2003

[Zimm08] Christian Zimmermann: *Implementierung einer Benutzerschnittstelle zur medizinischen Evaluierung eines Image Mosaicing-Verfahrens in*

der Endoskopie, Diplomarbeit, 2008, http://www.gm.fh-koeln.de/~konen/Diplom+Projekte/PaperPDF/DiplomZimmermann08.pdf, abgerufen am 22.11.2008

[ZF03] Barbara Zitova, Jan Flusser: *Image registration methods: a survey*, Department of Image Processing, Institute of Information Theory and Automation, Academy of Sciences of the Czech Republic, 2003, http://library.utia.cas.cz/prace/20030125.pdf, abgerufen am 28.11.2008

[ZFD97] I. Zoghlami, O. Faugeras, R. Deriche: *Using geometric corners to build a 2D mosaic from a set of images*, Computer Vision and Pattern Recognition, pp 421-425., 1997, ftp://ftp-sop.inria.fr/odyssee/Publications/1997/zoghlami-faugeras-etal:97.ps.gz, abgerufen am 28.11.2008

Anhang A

ImageJ-Plugin Parameter

MOTIONMODE = 1	# [0] 0: Proc.motion, 1: Proc.motion_log (LogSearch)
WARPCLASS = 1	# [1] 0: only translation, >0: full affine estimate of warping transformation
# Settings for panorama painting	
XMethod = 2	# [1] 1: new image paints only pixels which were not yet painted (X_Area is ring) # 2: new image paints all pixels it can anew (X_Area = union of old region & ring)
a_IIR = 4	# [4] 0: do not use IIR in diffImg, take plain difference # >0: subtract IIR-filtered image from each image before differencing
erode_rim = 8	# [0] erode the mask with a square of size erode_rim before using it in panorama painting # (avoid dark border region)
# Settings for LogSearch:	
log.rsz = 15	# [15] side length of correlation template square, must be odd # log.rsz may be 15 for 360x288 images, but should be 31-45 for 720x576 images
log.cross = 4	# [4] initial size of the search cross, must be power of 2
log.NPTS = 8	# [8] number of landmarks in each dimension, so the total # number of landmarks becomes roughly NPTS^2
log.cthresh = 0.85	# [0.85] minimum normalized correlation threshold; # landmarks with correlation below are discarded
log.uthresh = 2.5	# [2.5] maximum Euclidian distance between correlation # based motion vector and affine motion vector at a landmark; # landmarks with larger distance are discarded
log.pGuar = 0.2	# [0.2] accept anyway at least (log.pGuar*100)% of all landmarks

Tabelle A.1: Grundlegende Konfiguration der Parameter für das ImageJ-Plugin

Anhang B

CD-ROM

Ordner	Beschreibung
CUDA	CUDA 2.0 Installationsdateien und Installationsanleitung
Masterarbeit	Masterarbeit in elektronischer Form
Prototypen	Sämtliche CPU- und GPU-Prototypen des ImageJ-Plugins

Tabelle B.1: CD-ROM Inhalt

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zur haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Köln, den _____

_____
Eugen Sewergin